

Table of Contents

[Performance Center for SQL Server Database Engine and Azure SQL Database](#)

[Configuring Storage Spaces with a NVDIMM-N write-back cache](#)

[Adaptive Query Processing](#)

[Automatic Tuning](#)

[Cardinality Estimation](#)

[Monitor and Tune for Performance](#)

[Monitor SQL Server Components](#)

[Performance Monitoring and Tuning Tools](#)

[Establish a Performance Baseline](#)

[Isolate Performance Problems](#)

[Identify Bottlenecks](#)

[Server Performance and Activity Monitoring](#)

[Start System Monitor \(Windows\)](#)

[Set Up a SQL Server Database Alert \(Windows\)](#)

[View the Windows Application Log \(Windows\)](#)

[View the SQL Server Error Log \(SQL Server Management Studio\)](#)

[Save Deadlock Graphs \(SQL Server Profiler\)](#)

[Open, View, and Print a Deadlock File \(SQL Server Management Studio\)](#)

[Save Showplan XML Events Separately \(SQL Server Profiler\)](#)

[Save Showplan XML Statistics Profile Events Separately \(SQL Server Profiler\)](#)

[Display and Save Execution Plans](#)

[Display the Estimated Execution Plan](#)

[Display an Actual Execution Plan](#)

[Save an Execution Plan in XML Format](#)

[Live Query Statistics](#)

[Monitoring Performance By Using the Query Store](#)

[Using the Query Store with In-Memory OLTP](#)

[Best Practice with the Query Store](#)

[Query Store Usage Scenarios](#)

[How Query Store Collects Data](#)

[Database Tuning Advisor](#)

[Start and Use the Database Engine Tuning Advisor](#)

[Reduce the Production Server Tuning Load](#)

[Considerations for Using Test Servers](#)

[View and Work with the Output from the Database Engine Tuning Advisor](#)

[Columnstore index recommendations in Database Engine Tuning Advisor \(DTA\)](#)

[Tuning Database Using Workload from Query Store](#)

[Performance Improvements using Database Engine Tuning Advisor \(DTA\) recommendations](#)

[Plan Guides](#)

[Create a New Plan Guide](#)

[Create a Plan Guide for Parameterized Queries](#)

[Specify Query Parameterization Behavior by Using Plan Guides](#)

[Apply a Fixed Query Plan to a Plan Guide](#)

[Attach Query Hints to a Plan Guide](#)

[View Plan Guide Properties](#)

[Use SQL Server Profiler to Create and Test Plan Guides](#)

[Validate Plan Guides After Upgrade](#)





[Delete a Plan Guide](#)

[Enable or Disable a Plan Guide](#)

[Monitor Resource Usage \(System Monitor\)](#)


Performance Center for SQL Server Database Engine and Azure SQL Database

1/17/2018 • 2 min to read • [Edit Online](#)


THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

This page provides links to help you locate the information that you need about performance in the SQL Server Database Engine and Azure SQL Database.

Legend

 indicates that the feature is available only in SQL Server Database Engine (both SQL Server running on-premises and SQL Server running in an Azure Virtual Machine).

 indicates that the feature is available only in Azure SQL Database.








 indicates that the feature is available in both SQL Server and SQL Database.


















This is a work in process. Does this Performance Center help you? How can we improve it?

What information are you looking for, and did you find it? What are we missing? What would you like to see here? We're listening to your feedback to improve the content. Please submit your comments to sqlfeedback@microsoft.com

Configuration Options for Performance









SQL Server provides the ability to affect database engine performance through a number of configuration options at the SQL Server Database Engine level. With Azure SQL Database, Microsoft performs most, but not all, of these optimizations for you.

Disk configuration options	 Disk striping and RAID
Data and log file configuration options	 Place Data and Log Files on Separate Drives  View or Change the Default Locations for Data and Log Files (SQL Server Management Studio)
TempDB configuration options	 Performance Improvements in TempDB  Database Engine Configuration - TempDB  Using SSDs in Azure VMs to store SQL Server TempDB and Buffer Pool Extensions  Disk and performance best practices for temporary disk for SQL Server in Azure Virtual Machines

<p>Server Configuration Options</p>	<ul style="list-style-type: none"> • Processor configuration options <ul style="list-style-type: none"> ◦  affinity mask Server Configuration Option ◦  affinity Input-Output mask Server Configuration Option ◦  affinity64 mask Server Configuration Option ◦  affinity64 Input-Output mask Server Configuration Option ◦  Configure the max worker threads Server Configuration Option • Memory configuration options <ul style="list-style-type: none"> ◦  Server Memory Server Configuration Options • Index configuration options <ul style="list-style-type: none"> ◦  Configure the fill factor Server Configuration Option • Query configuration options <ul style="list-style-type: none"> ◦  Configure the min memory per query Server Configuration Option ◦  Configure the query governor cost limit Server Configuration Option ◦  Configure the max degree of parallelism Server Configuration Option ◦  Configure the cost threshold for parallelism Server Configuration Option ◦  optimize for ad hoc workloads Server Configuration Option • Backup configuration options <ul style="list-style-type: none"> ◦  View or Configure the backup compression default Server Configuration Option
<p>Database configuration optimization options</p>	<ul style="list-style-type: none">  Data Compression  View or Change the Compatibility Level of a Database  ALTER DATABASE SCOPED CONFIGURATION (Transact-SQL)
<p>Table configuration optimization</p>	<ul style="list-style-type: none">  Partitioned Tables and Indexes
<p>Database Engine Performance in an Azure Virtual Machine</p>	<ul style="list-style-type: none">  Quick check list  Virtual machine size and storage account considerations  Disks and performance considerations  I/O Performance Considerations  Feature specific performance considerations

Query Performance Options

--	--





 Indexes	Reorganize and Rebuild Indexes Specify Fill Factor for an Index Configure Parallel Index Operations SORT_IN_TEMPDB Option For Indexes Improve the Performance of Full-Text Indexes Configure the min memory per query Server Configuration Option Configure the index create memory Server Configuration Option
 Partitioned Tables and Indexes	Benefits of Partitioning
 Stored Procedures	CREATE PROCEDURE (Transact-SQL)
 User-Defined Functions	CREATE FUNCTION (Transact-SQL)
 Parallelism optimization	Configure the max worker threads Server Configuration Option ALTER DATABASE SCOPED CONFIGURATION (Transact-SQL)
 Query optimizer optimization	ALTER DATABASE SCOPED CONFIGURATION (Transact-SQL)
 Statistics	When to Update Statistics Update Statistics
 In-Memory OLTP (In-Memory Optimization)	Memory-Optimized Tables Natively Compiled Stored Procedures Creating and Accessing Tables in TempDB from Natively Compiled Stored Procedures Troubleshooting Common Performance Problems with Memory-Optimized Hash Indexes Demonstration: Performance Improvement of In-Memory OLTP

See Also

[Monitor and Tune for Performance](#)
[Monitoring Performance By Using the Query Store](#)
[Azure SQL Database performance guidance for single databases](#)
[Optimizing Azure SQL Database Performance using Elastic Pools](#)
[Azure Query Performance Insight](#)
[Index Design Guide](#)
[Memory Management Architecture Guide](#)
[Pages and Extents Architecture Guide](#)
[Post-migration Validation and Optimization Guide](#)
[Query Processing Architecture Guide](#)
[SQL Server Transaction Locking and Row Versioning Guide](#)
[SQL Server Transaction Log Architecture and Management Guide](#)
[Thread and Task Architecture Guide](#)

Configuring Storage Spaces with a NVDIMM-N write-back cache

1/17/2018 • 2 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

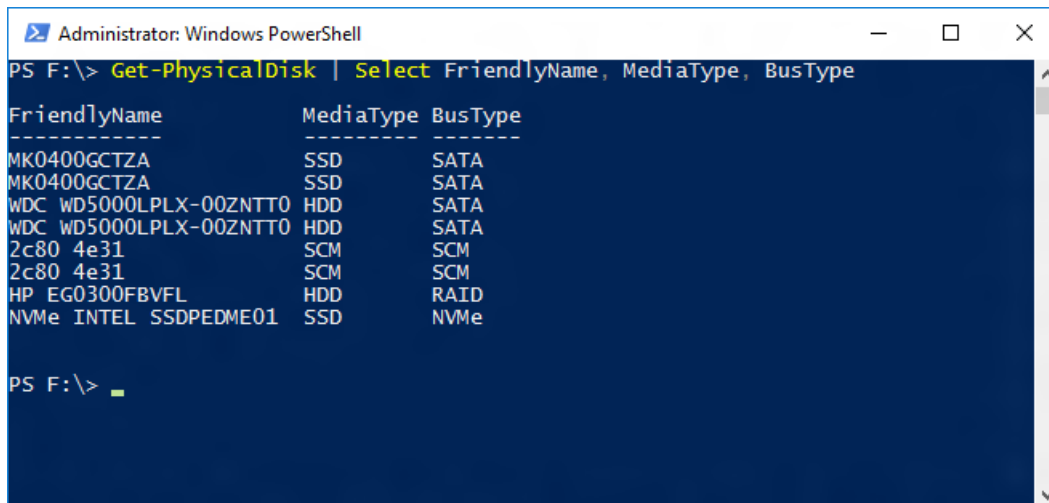
Windows Server 2016 supports NVDIMM-N devices that allow for extremely fast input/output (I/O) operations. One attractive way of using such devices is as a write-back cache to achieve low write latencies. This topic discusses how to set up a mirrored storage space with a mirrored NVDIMM-N write-back cache as a virtual drive to store the SQL Server transaction log. If you are looking to utilize it to also store data tables or other data, you may include more disks in the storage pool, or create multiple pools, if isolation is important.

To view a Channel 9 video using this technique, see [Using Non-volatile Memory \(NVDIMM-N\) as Block Storage in Windows Server 2016](#).

Identifying the right disks

Setup of storage spaces in Windows Server 2016, especially with advanced features, such as write-back caches is most easily achieved through PowerShell. The first step is to identify which disks should be part of the Storage Spaces pool that the virtual disk will be created from. NVDIMM-Ns have a media type and bus-type of SCM (storage class memory), which can be queried via the Get-PhysicalDisk PowerShell cmdlet.

```
Get-PhysicalDisk | Select FriendlyName, MediaType, BusType
```



```
Administrator: Windows PowerShell
PS F:\> Get-PhysicalDisk | Select FriendlyName, MediaType, BusType

FriendlyName      MediaType BusType
-----
MK0400GCTZA       SSD      SATA
MK0400GCTZA       SSD      SATA
WDC WD5000LPLX-00ZNTT0 HDD      SATA
WDC WD5000LPLX-00ZNTT0 HDD      SATA
2c80 4e31         SCM      SCM
2c80 4e31         SCM      SCM
HP EG0300FBVFL    HDD      RAID
NVMe INTEL SSDPEDME01 SSD      NVMe

PS F:\>
```

NOTE

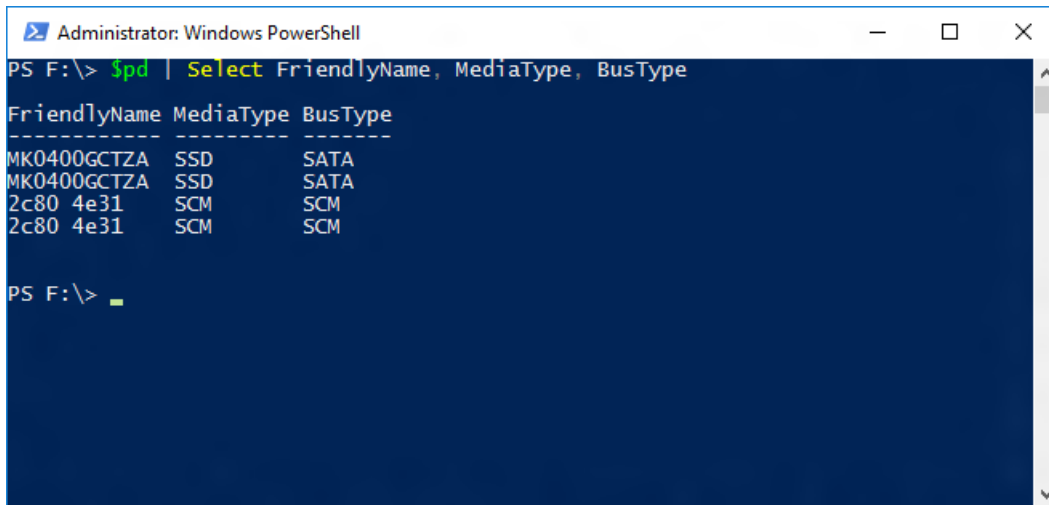
With NVDIMM-N devices, you no longer need to specifically select the devices that can be write-back cache targets.

In order to build a mirrored virtual disk with mirrored write-back cache, at least 2 NVDIMM-Ns, and 2 other disks are needed. Assigning the desired physical disks to a variable before building the pool makes the process easier.

```
$pd = Get-PhysicalDisk | Select FriendlyName, MediaType, BusType | Where-Object {$_.FriendlyName -like 'MK0*' -or $_.FriendlyName -like '2c80*'}
```

The screenshot shows the \$pd variable and the 2 SSDs and 2 NVDIMM-Ns it is assigned to returned using the following PowerShell cmdlet.

```
$pd | Select FriendlyName, MediaType, BusType
```



```
Administrator: Windows PowerShell
PS F:\> $pd | Select-FriendlyName, MediaType, BusType

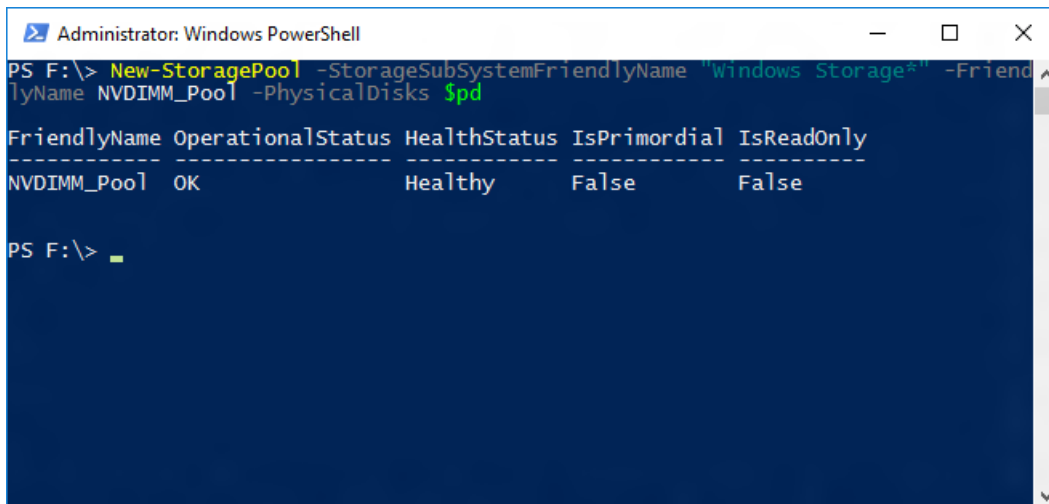
FriendlyName  MediaType  BusType
-----
MK0400GCTZA   SSD        SATA
MK0400GCTZA   SSD        SATA
2c80 4e31     SCM        SCM
2c80 4e31     SCM        SCM

PS F:\>
```

Creating the Storage Pool

Using the \$pd variable containing the PhysicalDisks, it is easy to build the storage pool using the New-StoragePool PowerShell cmdlet.

```
New-StoragePool -StorageSubSystemFriendlyName "Windows Storage*" -FriendlyName NVDIMM_Pool -PhysicalDisks $pd
```



```
Administrator: Windows PowerShell
PS F:\> New-StoragePool -StorageSubSystemFriendlyName "Windows Storage*" -FriendlyName NVDIMM_Pool -PhysicalDisks $pd

FriendlyName OperationalStatus HealthStatus IsPrimordial IsReadOnly
-----
NVDIMM_Pool OK Healthy False False

PS F:\>
```

Creating the Virtual Disk and Volume

Now that a pool has been created, the next step is to carve out a virtual disk and format it. In this case only 1 virtual disk will be created and the New-Volume PowerShell cmdlet can be used to streamline this process:

```
New-Volume -StoragePool (Get-StoragePool -FriendlyName NVDIMM_Pool) -FriendlyName Log_Space -Size 300GB -
FileSystem NTFS -AccessPath S: -ResiliencySettingName Mirror
```

```
Administrator: Windows PowerShell
PS F:\> New-Volume -StoragePool (Get-StoragePool -FriendlyName NVDIMM_Pool) -FriendlyName Log_Space -Size 300GB -FileSystem NTFS -AccessPath S: -ResiliencySettingName Mirror

DriveLetter FileSystemLabel FileSystem DriveType HealthStatus OperationalStatus
-----
S           Log_Space      NTFS      Fixed      Healthy      OK

PS F:\>
```

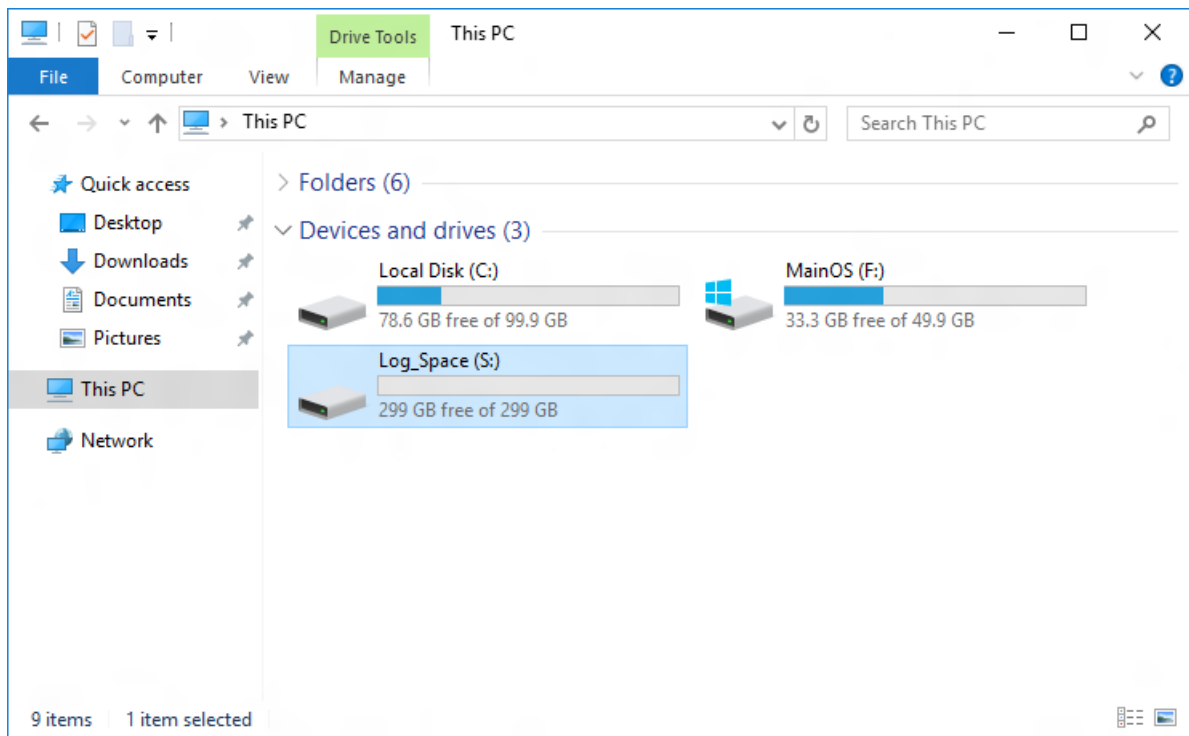
The virtual disk has been created, initialized, and formatted with NTFS. The screen capture below shows that it has a size of 300GB and a write-cache size of 1GB, which will be hosted on the NVDIMM-Ns.

```
Administrator: Windows PowerShell
PS F:\> Get-VirtualDisk | select FriendlyName,Size,WriteCacheSize,ResiliencySettingName

FriendlyName      Size WriteCacheSize ResiliencySettingName
-----
Log_Space      322122547200      1073741824 Mirror

PS F:\>
```

You can now view this new volume visible in your server. You can now use this drive for your SQL Server transaction log.



See Also

[Windows Storage Spaces in Windows 10](#)





[Windows Storage Spaces in Windows 2012 R2](#)

[The Transaction Log \(SQL Server\)](#)

[View or Change the Default Locations for Data and Log Files \(SQL Server Management Studio\)](#)

Adaptive query processing in SQL databases

1/17/2018 • 12 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

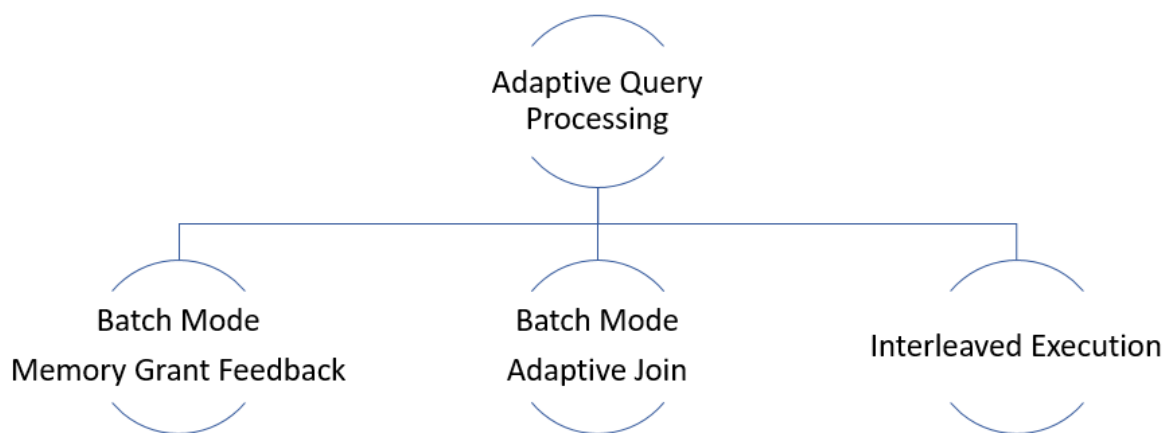
This article introduces these adaptive query processing features that you can use to improve query performance in SQL Server and Azure SQL Database:

- Batch mode memory grant feedback.
- Batch mode adaptive join.
- Interleaved execution.

At a general level, SQL Server executes a query as follows:

1. The query optimization process generates a set of feasible execution plans for a specific query. During this time, the cost of plan options is estimated and the plan with the lowest estimated cost is used.
2. The query execution process takes the plan chosen by the query optimizer and uses it for execution.

Sometimes the plan chosen by the query optimizer is not optimal for a variety of reasons. For example, the estimated number of rows flowing through the query plan may be incorrect. The estimated costs help determine which plan gets selected for use in execution. If cardinality estimates are incorrect, the original plan is still used despite the poor original assumptions.



How to enable adaptive query processing

You can make workloads automatically eligible for adaptive query processing by enabling compatibility level 140 for the database. You can set this using Transact-SQL. For example:

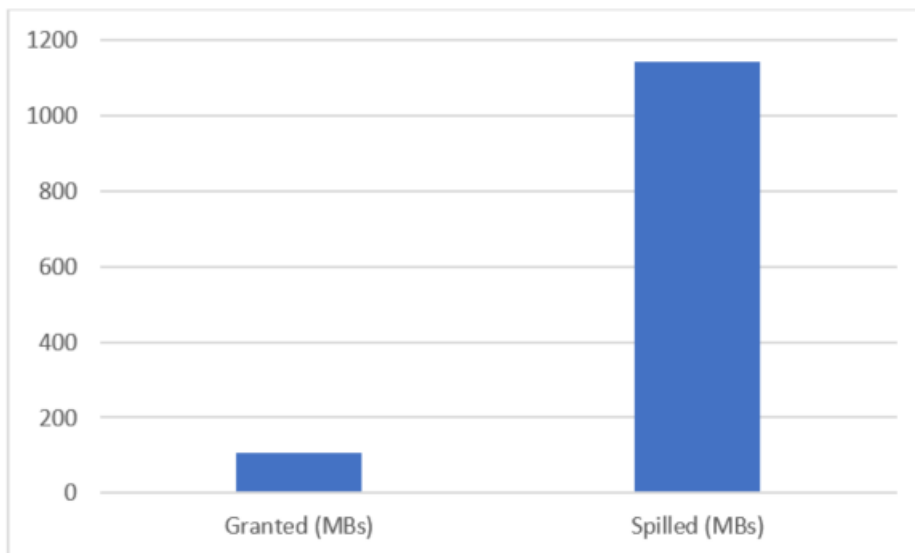
```
ALTER DATABASE [WideWorldImportersDW] SET COMPATIBILITY_LEVEL = 140;
```

Batch mode memory grant feedback

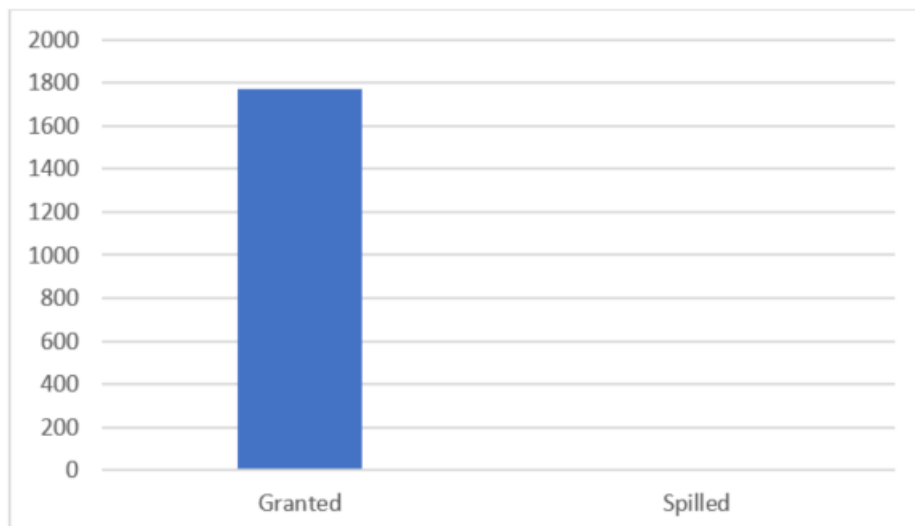
A query's post-execution plan in SQL Server includes the minimum required memory needed for execution and the ideal memory grant size to have all rows fit in memory. Performance suffers when memory grant sizes are incorrectly sized. Excessive grants result in wasted memory and reduced concurrency. Insufficient memory grants cause expensive spills to disk. By addressing repeating workloads, batch mode memory grant feedback recalculates the actual memory required for a query and then updates the grant value for the cached plan. When an identical

query statement is executed, the query uses the revised memory grant size, reducing excessive memory grants that impact concurrency and fixing underestimated memory grants that cause expensive spills to disk. The following graph shows one example of using batch mode adaptive memory grant feedback. For the first execution of the query, duration was *88 seconds* due to high spills:

```
DECLARE @EndTime datetime = '2016-09-22 00:00:00.000';
DECLARE @StartTime datetime = '2016-09-15 00:00:00.000';
SELECT TOP 10 hash_unique_bigint_id
FROM dbo.TelemetryDS
WHERE Timestamp BETWEEN @StartTime and @EndTime
GROUP BY hash_unique_bigint_id
ORDER BY MAX(max_elapsed_time_microsec) DESC;
```



With memory grant feedback enabled, for the second execution, duration is *1 second* (down from 88 seconds), spills are removed entirely, and the grant is higher:



Memory grant feedback sizing

For *excessive grants*, if the granted memory is more than two times the size of the actual used memory, memory grant feedback will recalculate the memory grant and update the cached plan. Plans with memory grants under 1 MB will not be recalculated for overages. For *insufficiently sized memory grants* that result in a spill to disk for batch mode operators, memory grant feedback will trigger a recalculation of the memory grant. Spill events are reported to memory grant feedback and can be surfaced via the *spilling_report_to_memory_grant_feedback* XEvent event. This event returns the node id from the plan and spilled data size of that node.

Memory grant feedback and parameter sensitive scenarios

Different parameter values may also require different query plans in order to remain optimal. This type of query is defined as “parameter-sensitive.” For parameter-sensitive plans, memory grant feedback will disable itself on a query if it has unstable memory requirements. The plan is disabled after several repeated runs of the query and this can be observed by monitoring the *memory_grant_feedback_loop_disabled* XEvent.

Memory grant feedback caching

Feedback can be stored in the cached plan for a single execution. It is the consecutive executions of that statement, however, that benefit from the memory grant feedback adjustments. This feature applies to repeated execution of statements. Memory grant feedback will change only the cached plan. Changes are currently not captured in the Query Store. Feedback is not persisted if the plan is evicted from cache. Feedback will also be lost if there is a failover. A statement using `OPTION(RECOMPILE)` creates a new plan and does not cache it. Since it is not cached, no memory grant feedback is produced and it is not stored for that compilation and execution. However, if an equivalent statement (that is, with the same query hash) that did *not* use `OPTION(RECOMPILE)` was cached and then re-executed, the consecutive statement can benefit from memory grant feedback.

Tracking memory grant feedback activity

You can track memory grant feedback events using the *memory_grant_updated_by_feedback* XEvent event. This event tracks the current execution count history, the number of times the plan has been updated by memory grant feedback, the ideal additional memory grant before modification and the ideal additional memory grant after memory grant feedback has modified the cached plan.

Memory grant feedback, resource governor and query hints

The actual memory granted honors the query memory limit determined by the resource governor or query hint.

Batch mode adaptive joins

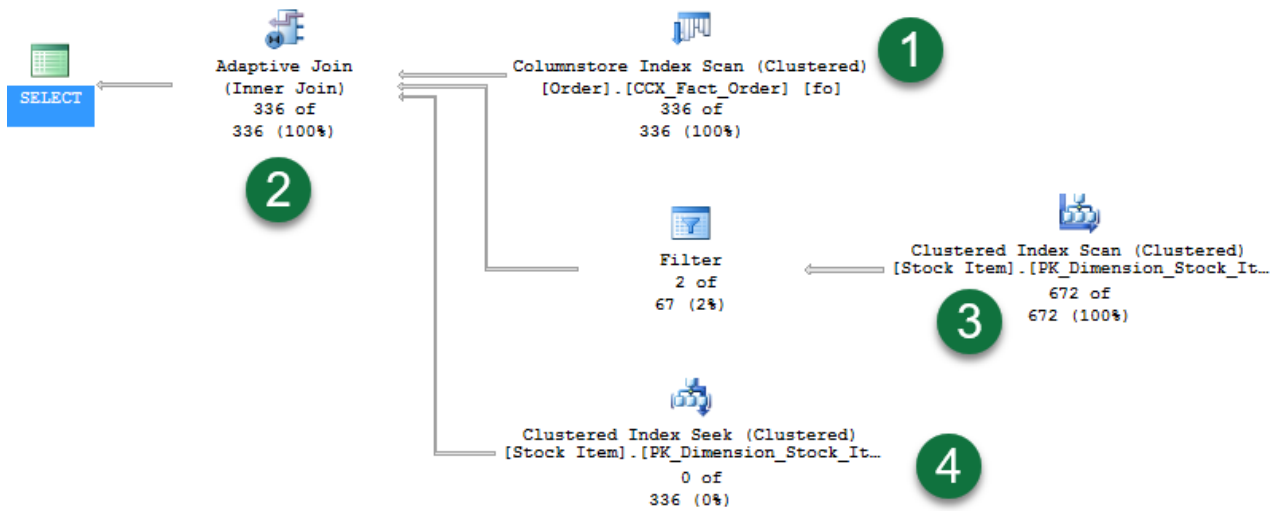
The batch mode adaptive joins feature enables the choice of a hash join or nested loop join method to be deferred until *after* the first input has been scanned. The adaptive join operator defines a threshold that is used to decide when to switch to a nested loop plan. Your plan can therefore dynamically switch to a better join strategy during execution. Here’s how it works:

- If the row count of the build join input is small enough that a nested loop join would be more optimal than a hash join, your plan switches to a nested loop algorithm.
- If the build join input exceeds a specific row count threshold, no switch occurs and your plan continues with a hash join.

The following query is used to illustrate an adaptive join example:

```
SELECT  [fo].[Order Key], [si].[Lead Time Days],
        [fo].[Quantity]
FROM    [Fact].[Order] AS [fo]
INNER JOIN [Dimension].[Stock Item] AS [si]
        ON [fo].[Stock Item Key] = [si].[Stock Item Key]
WHERE   [fo].[Quantity] = 360;
```

The query returns 336 rows. Enabling Live Query Statistics we see the following plan:

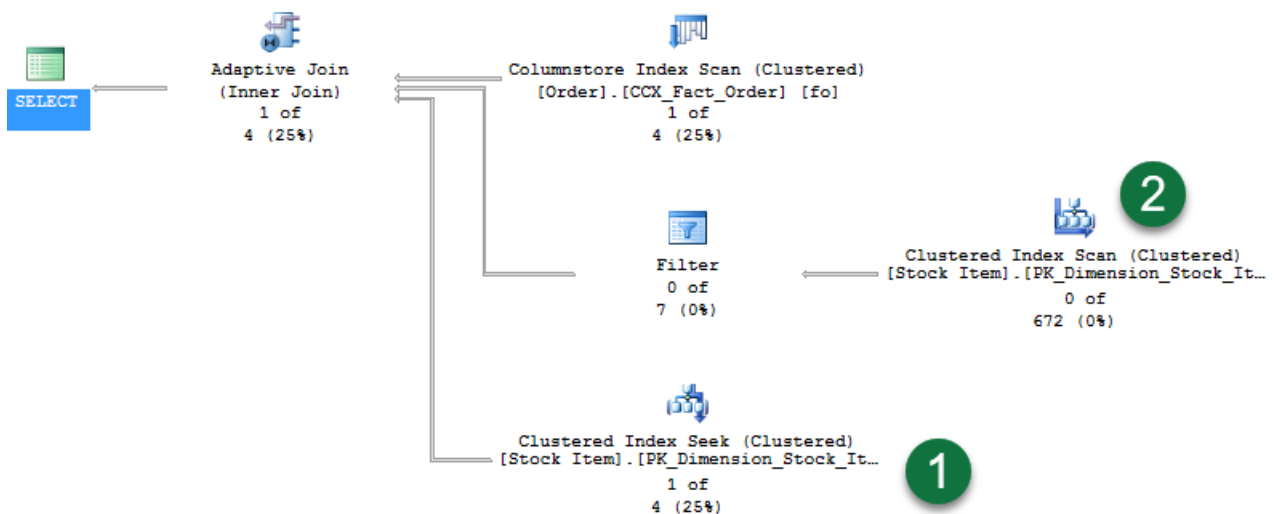


In the plan, we see the following:

1. We have a columnstore index scan used to provide rows for the hash join build phase.
2. We have the new adaptive join operator. This operator defines a threshold that is used to decide when to switch to a nested loop plan. For our example, the threshold is 78 rows. Anything with ≥ 78 rows will use a hash join. If less than the threshold, a nested loop join will be used.
3. Since we return 336 rows, we are exceeding the threshold and so the second branch represents the probe phase of a standard hash join operation. Notice that Live Query Statistics shows rows flowing through the operators – in this case “672 of 672”.
4. And the last branch is our Clustered Index Seek for use by the nested loop join had the threshold not been exceeded. Notice that we see “0 of 336” rows displayed (the branch is unused). Now let’s contrast the plan with the same query, but this time for a Quantity value that only has one row in the table:

```
SELECT [fo].[Order Key], [si].[Lead Time Days],
[fo].[Quantity]
FROM [Fact].[Order] AS [fo]
INNER JOIN [Dimension].[Stock Item] AS [si]
ON [fo].[Stock Item Key] = [si].[Stock Item Key]
WHERE [fo].[Quantity] = 361;
```

The query returns one row. Enabling Live Query Statistics we see the following plan:



In the plan, we see the following:

- With one row returned, you see the Clustered Index Seek now has rows flowing through it.
- And since we did not continue with the hash join build phase, you’ll see zero rows flowing through the second

branch.

Adaptive join benefits

Workloads with frequent oscillations between small and large join input scans will benefit most from this feature.

Adaptive join overhead

Adaptive joins introduce a higher memory requirement than an index nested loop join equivalent plan. The additional memory is requested as if the nested loop was a hash join. There is also overhead for the build phase as a stop-and-go operation versus a nested loop streaming equivalent join. With that additional cost comes flexibility for scenarios where row counts may fluctuate in the build input.

Adaptive join caching and re-use

Batch mode adaptive joins work for the initial execution of a statement, and once compiled, consecutive executions will remain adaptive based on the compiled adaptive join threshold and the runtime rows flowing through the build phase of the outer input.

Tracking adaptive join activity

The adaptive join operator has the following plan operator attributes:

PLAN ATTRIBUTE	DESCRIPTION
AdaptiveThresholdRows	Shows the threshold use to switch from a hash join to nested loop join.
EstimatedJoinType	What the join type is likely to be.
ActualJoinType	In an actual plan, shows what join algorithm was ultimately chosen based on the threshold.

The estimated plan shows the adaptive join plan shape, along with a defined adaptive join threshold and estimated join type.

Adaptive join and Query Store interoperability

Query Store captures and is able to force a batch mode adaptive join plan.

Adaptive join eligible statements

A few conditions make a logical join eligible for a batch mode adaptive join:

- The database compatibility level is 140
- The query is a SELECT statement (data modification statements are currently ineligible)
- The join is eligible to be executed both by an indexed nested loop join or a hash join physical algorithm
- The hash join uses batch mode – either through the presence of a Columnstore index in the query overall or a Columnstore indexed table being referenced directly by the join
- The generated alternative solutions of the nested loop join and hash join should have the same first child (outer reference)

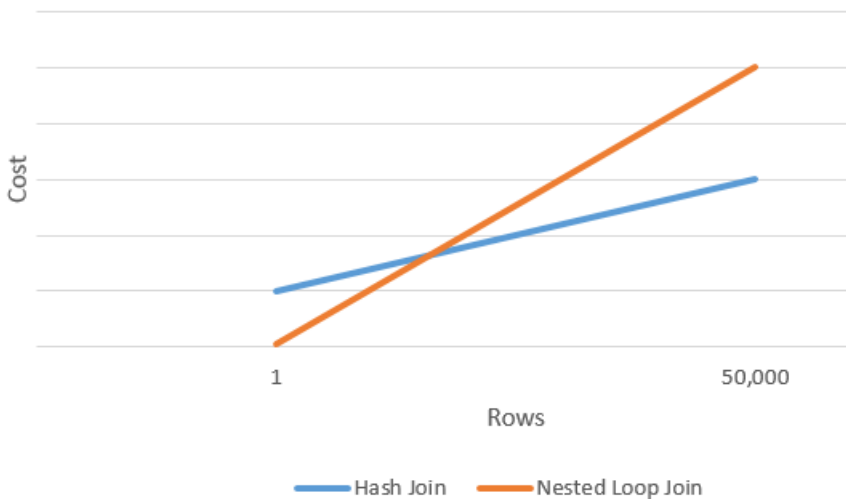
Adaptive joins and nested loop efficiency

If an adaptive join switches to a nested loop operation, it uses the rows already read by the hash join build. The operator does *not* re-read the outer reference rows again.

Adaptive threshold rows

The following chart shows an example intersection between the cost of a hash join vs. the cost of a nested loop join alternative. At this intersection point, the threshold is determined that in turn determines the actual algorithm used for the join operation.

Adaptive Join Threshold

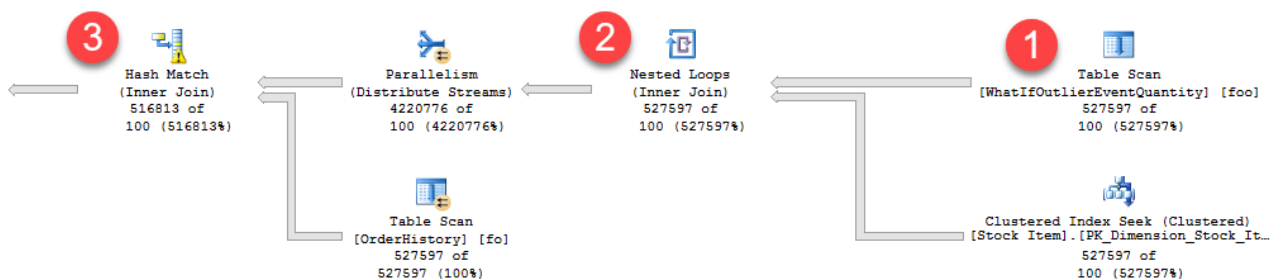


Interleaved execution for multi-statement table valued functions

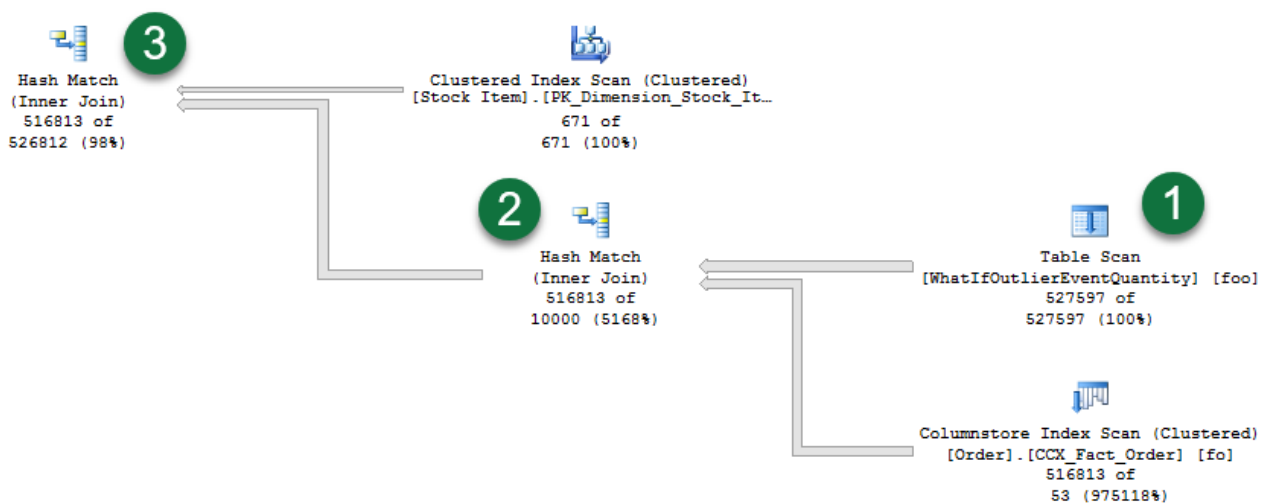
Interleaved execution changes the unidirectional boundary between the optimization and execution phases for a single-query execution and enables plans to adapt based on the revised cardinality estimates. During optimization if we encounter a candidate for interleaved execution, which is currently **multi-statement table valued functions (MSTVFs)**, we will pause optimization, execute the applicable subtree, capture accurate cardinality estimates, and then resume optimization for downstream operations. MSTVFs have a fixed cardinality guess of "100" in SQL Server 2014 and SQL Server 2016, and "1" for earlier versions. Interleaved execution helps workload performance issues that are due to these fixed cardinality estimates associated with multi-statement table valued functions.

The following image depicts a live query statistics output, a subset of an overall execution plan that shows the impact of fixed cardinality estimates from MSTVFs. You can see the actual row flow vs. estimated rows. There are three noteworthy areas of the plan (flow is from right to left):

1. The MSTVF Table Scan has a fixed estimate of 100 rows. For this example, however, there are 527,597 rows flowing through this MSTVF Table Scan as seen in Live Query Statistics via the "527597 of 100" actual of estimated – so the fixed estimate is significantly skewed.
2. For the Nested Loops operation, only 100 rows are assumed to be returned by the outer side of the join. Given the high number of rows actually being returned by the MSTVF, you are likely better off with a different join algorithm altogether.
3. For the Hash Match operation, notice the small warning symbol, which in this case is indicating a spill to disk.



Contrast the prior plan with the actual plan generated with interleaved execution enabled:



1. Notice that the MSTVF table scan now reflects an accurate cardinality estimate. Also notice the re-ordering of this table scan and the other operations.
2. And regarding join algorithms, we have switched from a Nested Loop operation to a Hash Match operation instead, which is more optimal given the large number of rows involved.
3. Also notice that we no longer have spill-warnings, as we're granting more memory based on the true row count flowing from the MSTVF table scan.

Interleaved execution eligible statements

MSTVF referencing statements in interleaved execution must currently be read-only and not part of a data modification operation. Also, MSTVFs are not eligible for interleaved execution if they do not use runtime constants.

Interleaved execution benefits

In general, the higher the skew between the estimated vs. actual number of rows, coupled with the number of downstream plan operations, the greater the performance impact. In general, interleaved execution benefits queries where:

1. There is a large skew between the estimated vs. actual number of rows for the intermediate result set (in this case, the MSTVF), and...
2. ...the overall query is sensitive to a change in the size of the intermediate result. This typically happens when there is a complex tree above that subtree in the query plan. A simply "SELECT *" from an MSTVF will not benefit from interleaved execution.

Interleaved execution overhead

The overhead should be minimal-to-none. MSTVFs were already being materialized prior to the introduction of interleaved execution, however the difference is that now we're now allowing deferred optimization and are then leveraging the cardinality estimate of the materialized row set. As with any plan affecting changes, some plans could change such that with better cardinality for the subtree we get a worse plan for the query overall. Mitigation can include reverting the compatibility level or using Query Store to force the non-regressed version of the plan.

Interleaved execution and consecutive executions

Once an interleaved execution plan is cached, the plan with the revised estimates on the first execution is used for consecutive executions without re-instantiating interleaved execution.

Tracking interleaved execution activity

You can see usage attributes in the actual query execution plan:

PLAN ATTRIBUTE	DESCRIPTION
----------------	-------------

PLAN ATTRIBUTE	DESCRIPTION
ContainsInterleavedExecutionCandidates	Applying to the <i>QueryPlan</i> node, when "true", it means the plan contains interleaved execution candidates.
IsInterleavedExecuted	The attribute is inside the RuntimeInformation element under the RelOp for the TVF node. When "true", it means the operation was materialized as part of an interleaved execution operation.

You can also track interleaved execution occurrences via the following XEvents:

XEVENT	DESCRIPTION
interleaved_exec_status	This event fires when interleaved execution is occurring.
interleaved_exec_stats_update	This event describes the cardinality estimates updated by interleaved execution.
Interleaved_exec_disabled_reason	This event fires when a query with a possible candidate for interleaved execution does not actually get interleaved execution.

A query must be executed in order to allow interleaved execution to revise MSTVF cardinality estimates. However, the estimated execution plan still shows when there are interleaved execution candidates via the ContainsInterleavedExecutionCandidates attribute.

Interleaved execution caching

If a plan is cleared or evicted from cache, upon query execution there is a fresh compilation that uses interleaved execution. A statement using OPTION(RECOMPILE) will create a new plan using interleaved execution and not cache it.

Interleaved execution and query store interoperability

Plans using interleaved execution can be forced. The plan is the version that has corrected cardinality estimates based on initial execution.

See Also

[Performance Center for SQL Server Database Engine and Azure SQL Database](#)

[Query Processing Architecture Guide](#)

[Demonstrating Adaptive Query Processing](#)

Automatic tuning

11/17/2017 • 11 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO: ✔ SQL Server (starting with 2017) ✔ Azure SQL Database ✖ Azure SQL Data Warehouse ✖ Parallel Data Warehouse

Automatic tuning is a database feature that provides insight into potential query performance problems, recommend solutions, and automatically fix identified problems.

Automatic tuning in SQL Server 2017 notifies you whenever a potential performance issue is detected, and lets you apply corrective actions, or lets the Database Engine automatically fix performance problems. Automatic tuning in SQL Server 2017 enables you to identify and fix performance issues caused by **SQL plan choice regressions**. Automatic tuning in Azure SQL Database creates necessary indexes and drops unused indexes.

Database Engine monitors the queries that are executed on the database and automatically improves performance of the workload. Database Engine has a built-in intelligence mechanism that can automatically tune and improve performance of your queries by dynamically adapting the database to your workload. There are two automatic tuning features that are available:

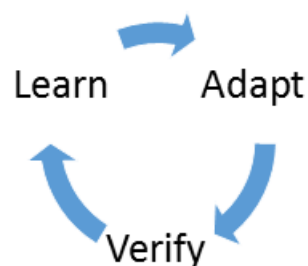
- **Automatic plan correction** (available in SQL Server 2017 and Azure SQL Database) that identifies problematic query execution plans and fixes SQL plan performance problems.
- **Automatic index management** (available only in Azure SQL Database) that identifies indexes that should be added in your database, and indexes that should be removed.

Why automatic tuning?

One of the main tasks in classic database administration is monitoring the workload, identifying critical Transact-SQL queries, indexes that should be added to improve performance, and rarely used indexes. Database Engine provides detailed insight into the queries and indexes that you need to monitor. However, constantly monitoring database is a hard and tedious task, especially when dealing with many databases. Managing a huge number of databases might be impossible to do efficiently. Instead of monitoring and tuning your database manually, you might consider delegating some of the monitoring and tuning actions to Database Engine using automatic tuning feature.

How does automatic tuning works?

Automatic tuning is a continuous monitoring and analysis process that constantly learns about the characteristic of your workload and identify potential issues and improvements.



This process enables database to dynamically adapt to your workload by finding what indexes and plans might improve performance of your workloads and what indexes affect your workloads. Based on these findings, automatic tuning applies tuning actions that improve performance of your workload. In addition, database continuously monitors performance after any change made by automatic tuning to ensure that it improves performance of your workload. Any action that didn't improve performance is automatically reverted. This

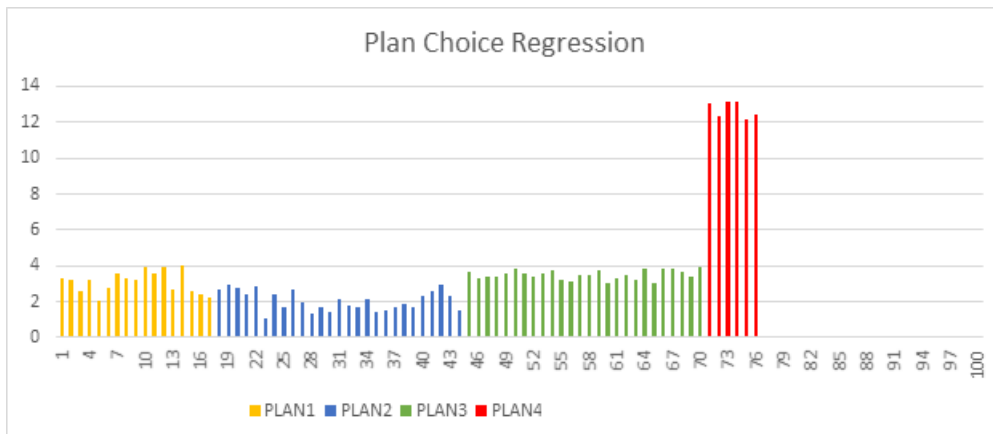
verification process is a key feature that ensures that any change made by automatic tuning does not decrease the performance of your workload.

Automatic plan correction

Automatic plan correction is an automatic tuning feature that identifies **SQL plans choice regression** and automatically fix the issue by forcing the last known good plan.

What is SQL plan choice regression?

SQL Server Database Engine may use different SQL plans to execute the Transact-SQL queries. Query plans depend on the statistics, indexes, and other factors. The optimal plan that should be used to execute some Transact-SQL query might be changed over time. In some cases, the new plan might not be better than the previous one, and the new plan might cause a performance regression.



Whenever you notice the plan choice regression, you should find some previous good plan and force it instead of the current one using `sp_query_store_force_plan` procedure. Database Engine in SQL Server 2017 provides information about regressed plans and recommended corrective actions. Additionally, Database Engine enables you to fully automate this process and let Database Engine fix any problem found related to the plan changes.

Automatic plan choice correction

Database Engine can automatically switch to the last known good plan whenever the plan choice regression is detected.



Database Engine automatically detects any potential plan choice regression including the plan that should be used instead of the wrong plan. When the Database Engine applies the last known good plan, it automatically monitors the performance of the forced plan. If the forced plan is not better than the regressed plan, the new plan will be unforced and the Database Engine will compile a new plan. If Database Engine verifies that the forced plan is better than regressed one, the forced plan will be retained until a recompile (for example, on next statistics or schema change) if it is better than the regressed plan.

Enabling automatic plan choice correction

You can enable automatic tuning per database and specify that last good plan should be forced whenever some plan change regression is detected. Automatic tuning is enabled using the following command:

```
ALTER DATABASE current
SET AUTOMATIC_TUNING ( FORCE_LAST_GOOD_PLAN = ON );
```

Once you turn-on this option, Database Engine will automatically force any recommendation where the estimated CPU gain is higher than 10 seconds, or the number of errors in the new plan is higher than the number of errors in the recommended plan, and verify that the forced plan is better than the current one.

Alternative - manual plan choice correction

Without automatic tuning, users must periodically monitor system and look for the queries that regressed. If any plan regressed, user should find some previous good plan and force it instead of the current one using `sp_query_store_force_plan` procedure. The best practice would be to force the last known good plan because older plans might be invalid due to statistic or index changes. The user who forces the last known good plan should monitor performance of the query that is executed using the forced plan and verify that forced plan works as expected. Depending on the results of monitoring and analysis, plan should be forced or user should find some other way to optimize the query. Manually forced plans should not be forced forever, because the Database Engine should be able to apply optimal plans. The user or DBA should eventually unforce the plan using `sp_query_store_unforce_plan` procedure, and let the Database Engine find the optimal plan.

SQL Server provides all necessary views and procedures required to monitor performance and fix problems in Query Store.

In SQL Server 2016, you can find plan choice regressions using Query Store system views. In SQL Server 2017, the Database Engine detects and shows potential plan choice regressions and the recommended actions that should be applied in the [sys.dm_db_tuning_recommendations \(Transact-SQL\)](#) view. The view shows information about the problem, the importance of the issue, and details such as the identified query, the ID of the regressed plan, the ID of the plan that was used as baseline for comparison, and the Transact-SQL statement that can be executed to fix the problem.

TYPE	DESCRIPTION	DATETIME	SCORE	DETAILS	...
FORCE_LAST_GOOD_PLAN	CPU time changed from 4 ms to 14 ms	3/17/2017	83	queryId recommendedPlanId regressedPlanId T-SQL	
FORCE_LAST_GOOD_PLAN	CPU time changed from 37 ms to 84 ms	3/16/2017	26	queryId recommendedPlanId regressedPlanId T-SQL	

Some columns from this view are described in the following list:

- Type of the recommended action - `FORCE_LAST_GOOD_PLAN`.
- Description that contains information why Database Engine thinks that this plan change is a potential performance regression.
- Datetime when the potential regression is detected.
- Score of this recommendation.
- Details about the issues such as ID of the detected plan, ID of the regressed plan, ID of the plan that should be forced to fix the issue, Transact-SQL script that might be applied to fix the issue, etc. Details are stored in [JSON format](#).

Use the following query to obtain a script that fixes the issue and additional information about the estimated gain:

```
SELECT reason, score,
       script = JSON_VALUE(details, '$.implementationDetails.script'),
       planForceDetails.*,
       estimated_gain = (regressedPlanExecutionCount+recommendedPlanExecutionCount)
                       *(regressedPlanCpuTimeAverage-recommendedPlanCpuTimeAverage)/1000000,
       error_prone = IIF(regressedPlanErrorCount>recommendedPlanErrorCount, 'YES', 'NO')
FROM sys.dm_db_tuning_recommendations
CROSS APPLY OPENJSON (Details, '$.planForceDetails')
WITH ( [query_id] int '$.queryId',
       [current plan_id] int '$.regressedPlanId',
       [recommended plan_id] int '$.recommendedPlanId',

       regressedPlanErrorCount int,
       recommendedPlanErrorCount int,

       regressedPlanExecutionCount int,
       regressedPlanCpuTimeAverage float,
       recommendedPlanExecutionCount int,
       recommendedPlanCpuTimeAverage float

       ) as planForceDetails;
```

Here is the result set.

REASON	SCORE	SCRIPT	QUERY_ID	CURRENT PLAN_ID	RECOMMENDED PLAN_ID	ESTIMATED_GAIN	ERROR_PRONE
CPU time changed from 3 ms to 46 ms	36	EXEC sp_query_store_force_plan 12, 17;	12	28	17	11.59	0

`estimated_gain` represents the estimated number of seconds that would be saved if the recommended plan would be executed instead of the current plan. The recommended plan should be forced instead of the current plan if the gain is greater than 10 seconds. If there are more errors (for example, time-outs or aborted executions) in the current plan than in the recommended plan, the column `error_prone` would be set to the value `YES`. Error prone plan is another reason why the recommended plan should be forced instead of the current one.

Although Database Engine provides all information required to identify plan choice regressions; continuous monitoring and fixing performance issues might be a tedious process. Automatic tuning makes this process much easier.

Automatic index management

In Azure SQL Database, index management is easy because Azure SQL Database learns about your workload and ensures that your data is always optimally indexed. Proper index design is crucial for optimal performance of your workload, and automatic index management can help you optimize your indexes. Automatic index management can either fix performance issues in incorrectly indexed databases, or maintain and improve indexes on the existing database schema. Automatic tuning in Azure SQL Database performs the following actions:

- Identifies indexes that could improve performance of your T-SQL queries that read data from the tables.
- Identifies the redundant indexes or indexes that were not used in longer period of time that could be removed. Removing unnecessary indexes improves performance of the queries that update data in tables.

Why do you need index management?

Indexes speed up some of your queries that read data from the tables; however, they can slow down the queries that update data. You need to carefully analyze when to create an index and what columns you need to include in

the index. Some indexes might not be needed after some time. Therefore, you would need to periodically identify and drop the indexes that do not bring any benefits. If you ignore the unused indexes, performance of the queries that update data would be decreased without any benefit on the queries that read data. Unused indexes also affect overall performance of the system because additional updates require unnecessary logging.

Finding the optimal set of indexes that improve performance of the queries that read data from your tables and have minimal impact on updates might require continuous and complex analysis.

Azure SQL Database uses built-in intelligence and advanced rules that analyze your queries, identify indexes that would be optimal for your current workloads, and the indexes might be removed. Azure SQL Database ensures that you have a minimal necessary set of indexes that optimize the queries that read data, with the minimized impact on the other queries.

Automatic index management

In addition to detection, Azure SQL Database can automatically apply identified recommendations. If you find that the built-in rules improve the performance of your database, you might let Azure SQL Database automatically manage your indexes.

To enable automatic tuning in Azure SQL Database and let automatic tuning feature fully manage your workload, see [Enable automatic tuning in Azure SQL Database using Azure portal](#).

When the Azure SQL Database applies a CREATE INDEX or DROP INDEX recommendation, it automatically monitors the performance of the queries that are affected by the index. New index will be retained only if performances of the affected queries are improved. Dropped index will be automatically re-created if there are some queries that run slower due to the absence of the index.

Automatic index management considerations

Actions required to create necessary indexes in Azure SQL Database might consume resources and temporally affect workload performance. To minimize the impact of index creation on workload performance, Azure SQL Database finds the appropriate time window for any index management operation. Tuning action is postponed if the database needs resources to execute your workload, and started when the database has enough unused resources that can be used for the maintenance task. One important feature in automatic index management is a verification of the actions. When Azure SQL Database creates or drops index, a monitoring process analyzes performance of your workload to verify that the action improved the performance. If it didn't bring significant improvement – the action is immediately reverted. This way, Azure SQL Database ensures that automatic actions do not negatively impact performance of your workload. Indexes created by automatic tuning are transparent for the maintenance operation on the underlying schema. Schema changes such as dropping or renaming columns are not blocked by the presence of automatically created indexes. Indexes that are automatically created by Azure SQL Database are immediately dropped when related table or columns is dropped.

Alternative - manual index management

Without automatic index management, user would need to manually query [sys.dm_db_missing_index_details \(Transact-SQL\)](#) view to find indexes that might improve performance, create indexes using the details provided in this view, and manually monitor performance of the query. In order to find the indexes that should be dropped, users should monitor operational usage statistics of the indexes to find rarely used indexes.

Azure SQL Database simplifies this process. Azure SQL Database analyzes your workload, identifies the queries that could be executed faster with a new index, and identifies unused or duplicated indexes. Find more information about identification of indexes that should be changed at [Find index recommendations in Azure portal](#).





See Also

[ALTER DATABASE SET AUTOMATIC_TUNING \(Transact-SQL\)](#)
[sys.database_automatic_tuning_options \(Transact-SQL\)](#)
[sys.dm_db_tuning_recommendations \(Transact-SQL\)](#)

[sys.dm_db_missing_index_details \(Transact-SQL\)](#)
[sp_query_store_force_plan \(Transact-SQL\)](#)
[sp_query_store_unforce_plan \(Transact-SQL\)](#)
[sys.database_query_store_options \(Transact-SQL\)](#)
[JSON functions](#)

Cardinality Estimation (SQL Server)

1/17/2018 • 7 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

This article illustrates how you can assess and choose the best cardinality estimation (CE) configuration for your SQL system. Most systems benefit from the latest CE because it is the most accurate. The CE predicts how many rows your query will likely return. The cardinality prediction is used by the Query Optimizer to generate the optimal query plan. With more accurate estimations, the Query Optimizer can usually do a better job of producing a more optimal query plan.

Your application system could possibly have an important query whose plan is changed to a slower plan due to the new CE. Such a query might be like one of the following:

- An OLTP (online transaction processing) query that runs so frequently that multiple instance of it often run concurrently.
- A SELECT with substantial aggregation that runs during your OLTP business hours.

You have techniques for identifying a query that performs slower with the new CE. And you have options for how to address the performance issue.

Versions of the CE

In 1998, a major update of the CE was part of Microsoft SQL Server 7.0, for which the compatibility level was 70. Subsequent updates started with SQL Server 2014, meaning compatibility levels 120 and above. The CE updates for levels 120 and above incorporate assumptions and algorithms that work well on modern data warehousing and on OLTP workloads.

Compatibility level: You can ensure your database is at a particular level by using the following Transact-SQL code for [COMPATIBILITY_LEVEL](#).

```
SELECT ServerProperty('ProductVersion');
go

ALTER DATABASE <yourDatabase>
    SET COMPATIBILITY_LEVEL = 130;
go

SELECT d.name, d.compatibility_level
FROM sys.databases AS d
WHERE d.name = 'yourDatabase';
go
```

For a SQL Server database set at compatibility level 120 or above, activation of the [trace flag 9481](#) forces the system to use the CE version 70.

Legacy CE: For a SQL Server database set at compatibility level 120 and above, the CE version 70 can be activated by using the at the database level by using the [ALTER DATABASE SCOPED CONFIGURATION](#).


```

ALTER DATABASE
    SCOPED CONFIGURATION
        SET LEGACY_CARDINALITY_ESTIMATION = ON;

go

SELECT name, value
    FROM sys.database_scoped_configurations
    WHERE name = 'LEGACY_CARDINALITY_ESTIMATION';

```

Or starting with SQL Server 2016 SP1, the [Query Hint](#) `USE HINT ('FORCE_LEGACY_CARDINALITY_ESTIMATION')`.

```

SELECT CustomerId, OrderAddedDate
    FROM OrderTable
    WHERE OrderAddedDate >= '2016-05-01';
OPTION (USE HINT ('FORCE_LEGACY_CARDINALITY_ESTIMATION'));

```

Query store: Starting with SQL Server 2016, the query store is a handy tool for examining the performance of your queries. In Management Studio, in the **Object Explorer** under your database node, a **Query Store** node is displayed when the query store is enabled.

```

ALTER DATABASE <yourDatabase>
    SET QUERY_STORE = ON;
go

SELECT
    q.actual_state_desc AS [actual_state_desc-ofQueryStore],
    q.desired_state_desc,
    q.query_capture_mode_desc
    FROM
        sys.database_query_store_options AS q;
go

ALTER DATABASE <yourDatabase>
    SET QUERY_STORE CLEAR;

```

TIP

We recommend that you install the latest release of [Management Studio](#) and update it often.

Another option for tracking the cardinality estimation process is to use the extended event named **query_optimizer_estimate_cardinality**. The following T-SQL code sample runs on SQL Server. It writes a .xel file to C:\Temp\ (although you can change the path). When you open the .xel file in Management Studio, its detailed information is displayed in a user friendly manner.

```

DROP EVENT SESSION Test_the_CE_qoec_1 ON SERVER;
go

CREATE EVENT SESSION Test_the_CE_qoec_1
ON SERVER
ADD EVENT sqlserver.query_optimizer_estimate_cardinality
(
    ACTION (sqlserver.sql_text)
    WHERE (
        sql_text LIKE '%yourTable%'
        and sql_text LIKE '%SUM%'
    )
)
ADD TARGET package0.asynchronous_file_target
(SET
    filename = 'c:\temp\xe_qoec_1.xel',
    metadatafile = 'c:\temp\xe_qoec_1.xem'
);
go

ALTER EVENT SESSION Test_the_CE_qoec_1
ON SERVER
STATE = START; --STOP;
go

```

For information about extended events as tailored for SQL Database, see [Extended events in SQL Database](#).

Steps to assess the CE version

Next are steps you can use to assess whether any of your most important queries perform less well under the latest CE. Some of the steps are performed by running a code sample presented in a preceding section.

1. Open Management Studio. Ensure your SQL Server database is set to the highest available compatibility level.
2. Perform the following preliminary steps:
 - a. Open Management Studio.
 - b. Run the T-SQL to ensure that your SQL Server database is set to the highest available compatibility level.
 - c. Ensure that your database has its `LEGACY_CARDINALITY_ESTIMATION` configuration turned OFF.
 - d. CLEAR your query store. Of course, ensure your query store is ON.
 - e. Run the statement: `SET NOCOUNT OFF;`
3. Run the statement: `SET STATISTICS XML ON;`
4. Run your important query.
5. In the results pane, on the **Messages** tab, note the actual number of rows affected.
6. In the results pane on the **Results** tab, double-click the cell that contains the statistics in XML format. A graphic query plan is displayed.
7. Right-click the first box in the graphic query plan, and then click **Properties**.
8. For later comparison with a different configuration, note the values for the following properties:
 - **CardinalityEstimationModelVersion**.

- **Estimated Number of Rows.**
 - **Estimated I/O Cost**, and several similar *Estimated* properties that involve actual performance rather than row count predictions.
 - **Logical Operation** and **Physical Operation**. *Parallelism* is a good value.
 - **Actual Execution Mode**. *Batch* is a good value, better than *Row*.
9. Compare the estimated number of rows to the actual number of rows. Is the CE inaccurate by 1% (high or low), or by 10%?
 10. Run: `SET STATISTICS XML OFF;`
 11. Run the T-SQL to decrease the compatibility level of your database by one level (such as from 130 down to 120).
 12. Rerun all the non-preliminary steps.
 13. Compare the CE property values from the two runs.
 - Is the inaccuracy percentage under the newest CE less than under the older CE?
 14. Finally, compare the various performance property values from the two runs.
 - Did your query use a different plan under the two differing CE estimations?
 - Did your query run slower under the latest CE?
 - Unless your query runs better and with a different plan under the older CE, you almost certainly want the latest CE.
 - However, if your query runs with a faster plan under the older CE, consider forcing the system to use the faster plan and to ignore the CE. This way you can have the latest CE on for everything, while keeping the faster plan in the one odd case.

How to activate the best query plan

Suppose that with the new CE a slower query plan is generated for your query. Here are some options you have to activate the faster plan.

You could set the compatibility level to a value lower than the latest available, for your whole database.

- This activates the Legacy CE, but it makes all queries subject to the older and less accurate CE.
- Further the previous level compatibility also loses excellent improvements in the query optimizer.

You could use `LEGACY_CARDINALITY_ESTIMATION` to have the whole database use the older CE, or just a specific query, while retaining the improvements in the query optimizer.

For the finest control, you could *force* the SQL system to use the plan that was generated with the older CE during your testing. After you *pin* your preferred plan, you can set your whole database to use the latest compatibility level and CE. The option is elaborated next.

How to force a particular query plan

The query store gives you different ways that you can force the system to use a particular query plan:

- Execute **sp_query_store_force_plan**.
- In Management Studio, expand your **Query Store** node, right-click **Top Resource Consuming Nodes**, and then click **View Top Resource Consuming Nodes**. The display shows buttons labeled **Force Plan** and **Unforce Plan**.

For more information about the query store, see [Monitoring Performance By Using the Query Store](#).

Examples of CE improvements

This section describes example queries that benefit from the enhancements implemented in the CE in recent releases. This is background information that does not call for specific action on your part.

Example A. CE understands maximum value might be higher than when statistics were last gathered

Suppose statistics were last gathered for OrderTable on 2016-04-30, when the maximum OrderAddedDate was 2016-04-30. The CE for compatibility level 120 (and for higher levels) understands that columns in OrderTable which have *ascending* data might have values larger than the maximum recorded by the statistics. This understanding improves the query plan for SQL SELECTs such as the following.

```
SELECT CustomerId, OrderAddedDate
FROM OrderTable
WHERE OrderAddedDate >= '2016-05-01';
```

Example B. CE understands that filtered predicates on the same table are often correlated

In the following SELECT we see filtered predicates on Model and ModelVariant. We intuitively understand that when Model is 'Xbox' there is a chance the ModelVariant is 'One', given that Xbox has a variant called One.

The level 120 CE understands there might be a correlation between the two columns on the same table, Model and ModelVariant. The CE makes a more accurate estimation of how many rows will be returned by the query, and the query optimizer generates a more optimal plan.

```
SELECT Model, Purchase_Price
FROM dbo.Hardware
WHERE
    Model = 'Xbox' AND
    ModelVariant = 'One';
```

Example C. CE no longer assumes any correlation between filtered predicates from different tables

With extensive new research on modern workloads and actual business data reveal that predicate filters from different tables usually do not correlate with each other. In the following query, the CE assumes there is no correlation between s.type and r.date. Therefore the CE makes a lower estimate of the number of rows returned.

```
SELECT s.ticket, s.customer, r.store
FROM
    dbo.Sales AS s
CROSS JOIN dbo>Returns AS r
WHERE
    s.ticket = r.ticket AND
    s.type = 'toy' AND
    r.date = '2016-05-11';
```

See Also

[Monitor and Tune for Performance](#)

[Optimizing Your Query Plans with the SQL Server 2014 Cardinality Estimator](#)





[Query Hints](#)

[Monitoring Performance By Using the Query Store](#)

[Query Processing Architecture Guide](#)

Monitor and Tune for Performance

1/17/2018 • 3 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

The goal of monitoring databases is to assess how a server is performing. Effective monitoring involves taking periodic snapshots of current performance to isolate processes that are causing problems, and gathering data continuously over time to track performance trends.

Ongoing evaluation of the database performance helps you minimize response times and maximize throughput, yielding optimal performance. Efficient network traffic, disk I/O, and CPU usage are key to peak performance. You need to thoroughly analyze the application requirements, understand the logical and physical structure of the data, assess database usage, and negotiate tradeoffs between conflicting uses such as online transaction processing (OLTP) versus decision support.

Monitoring and tuning databases for performance

Microsoft SQL Server and the Microsoft Windows operating system provide utilities to view the current condition of the database and track performance as conditions change. There are a variety of tools and techniques you can use to monitor Microsoft SQL Server. Monitoring SQL Server helps you:

- Determine whether you can improve performance. For example, by monitoring the response times for frequently used queries, you can determine whether changes to the query or indexes on the tables are required.
- Evaluate user activity. For example, by monitoring users trying to connect to an instance of SQL Server, you can determine whether security is set up adequately and test applications or development systems. For example, by monitoring SQL queries as they are executed, you can determine whether they are written correctly and producing the expected results.
- Troubleshoot problems or debug application components, such as stored procedures.

Monitoring in a dynamic environment

Changing conditions result in changing performance. In your evaluations, you can see performance changes as the number of users increases, user access and connection methods change, database contents grow, client applications change, data in the applications changes, queries become more complex, and network traffic rises. Using tools to monitor performance helps you associate changes in performance with changing conditions and complex queries. **Examples::**

- By monitoring the response times for frequently used queries, you can determine whether changes to the query or indexes on the tables where the queries execute are required.
- By monitoring Transact-SQL queries as they are executed, you can determine whether the queries are written correctly and producing the expected results.
- By monitoring users that try to connect to an instance of SQL Server, you can determine whether security is set up adequately and test applications or development systems.

Response time is the length of time required for the first row of the result set to be returned to the user in the form of visual confirmation that a query is being processed. Throughput is the total number of queries handled by the server during a specified period of time.

As the number of users increases, so does the competition for a server's resources, which in turn increases response time and decreases overall throughput.

Monitoring and performance tuning tasks

TOPIC	TASK
Monitor SQL Server Components	Required steps to monitor any SQL Server component.
Performance Monitoring and Tuning Tools	Lists the monitoring and tuning tools available with SQL Server.
Establish a Performance Baseline	How to establish a performance baseline.
Isolate Performance Problems	Isolate database performance problems.
Identify Bottlenecks	Monitor and track server performance to identify bottlenecks.
Server Performance and Activity Monitoring	Use SQL Server and Windows performance and activity monitoring tools.
Display and Save Execution Plans	Display and save execution plans to a file in XML format.
Live Query Statistics	Display real-time statistics about query execution steps.
Monitoring Performance By Using the Query Store	Use Query Store to automatically capture a history of queries, plans, and runtime statistics, and retain these for your review.
Using the Query Store with In-Memory OLTP	Considerations for Memory-Optimized tables.
Best Practice with the Query Store	Advice on using the Query Store.

See also

[Automated Administration Across an Enterprise](#)





[Database Engine Tuning Advisor](#)

[Monitor Resource Usage \(System Monitor\)](#)

[SQL Server Profiler](#)

Monitor SQL Server Components

1/17/2018 • 5 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

Monitoring is important because SQL Server provides a service in a dynamic environment. The data in the application changes. The type of access that users require changes. The way that users connect changes. The types of applications accessing SQL Server may even change, but SQL Server automatically manages system-level resources, such as memory and disk space, to minimize the need for extensive system-level manual tuning. Monitoring lets administrators identify performance trends to determine if changes are necessary.

To monitor any component of SQL Server effectively:

1. Determine your monitoring goals.
2. Select the appropriate tool.
3. Identify components to monitor.
4. Select metrics for those components.
5. Monitor the server.
6. Analyze the data.

These steps are discussed in turn below.

Determine Your Monitoring Goals

To monitor SQL Server effectively you should clearly identify your reason for monitoring. Reasons can include the following:

- Establish a baseline for performance.
- Identify performance changes over time.
- Diagnose specific performance problems.
- Identify components or processes to optimize.
- Compare the effect of different client applications on performance.
- Audit user activity.
- Test a server under different loads.
- Test database architecture.
- Test maintenance schedules.
- Test backup and restore plans.
- Determining when to modify your hardware configuration.

Select the Appropriate Tool

After determining why you are monitoring, you should select the appropriate tools for that type of monitoring. The

Windows operating system and SQL Server provide a complete set of tools to monitor servers in transaction-intensive environments. These tools clearly reveal the condition of an instance of the SQL Server Database Engine or an instance of SQL Server Analysis Services.

Windows provides the following tools for monitoring applications that are running on a server:

- System Monitor, which lets you collect and view real-time data about activities such as memory, disk, and processor usage
- Performance logs and alerts
- Task Manager

For more information about Windows Server or Windows tools, see the [Windows documentation](#).

SQL Server provides the following tools for monitoring components of SQL Server:

- SQL Trace
- SQL Server Profiler
- Distributed Replay Utility
- SQL Server Management Studio Activity Monitor
- SQL Server Management Studio Graphical Showplan
- Stored procedures
- Database Console Commands (DBCC)
- Built-in functions
- Trace flags

For more information about SQL Server monitoring tools, see [Performance Monitoring and Tuning Tools](#).

Identify the Components to Monitor

The third step to monitoring an instance of SQL Server is to identify the components that you monitor. For example, if you are using SQL Server Profiler to trace a server you can define the trace to collect data about specific events. You can also exclude events that do not apply to your situation.

Select Metrics for Monitored Components

After identifying the components to monitor, determine the metrics for components you monitor. For example, after selecting the events to include in a trace, you can choose to include only specific data about the events. Limiting the trace to data that is relevant to the trace minimizes the system resources required to perform the tracing.

Monitor the Server

To monitor the server, run the monitoring tool that you have configured to gather data. For example, after a trace is defined, you can run the trace to gather data about events raised in the server.

Analyze the Data

After the trace has finished, analyze the data to see if you have achieved your monitoring goal. If you have not, modify the components or metrics that you used to monitor the server.

The following outlines the process for capturing event data and putting it to use.

1. Apply filters to limit the event data collected.

Limiting the event data allows for the system to focus on the events pertinent to the monitoring scenario. For example, if you want to monitor slow queries, you can use a filter to monitor only those queries issued by the application that take more than 30 seconds to run against a particular database. For more information, see [Set a Trace Filter \(Transact-SQL\)](#) and [Filter Events in a Trace \(SQL Server Profiler\)](#).

2. Monitor (capture) events.

As soon as it is enabled, active monitoring captures data from the specified application, instance of SQL Server, or operating system. For example, when disk activity is monitored using System Monitor, monitoring captures event data, such as disk reads and writes, and displays it on the screen. For more information, see [Monitor Resource Usage \(System Monitor\)](#).

3. Save captured event data.

Saving captured event data lets you analyze it later or even replay it using the Distributed Replay Utility or SQL Server Profiler. Captured event data is saved to a file that can be loaded back into the tool that originally created it for analysis. SQL Server Profiler permits event data to be saved to a SQL Server table. Saving captured event data is important when you are creating a performance baseline. The performance baseline data is saved and used, when comparing recently captured event data, to determine whether performance is optimal. For more information, see [SQL Server Profiler Templates and Permissions](#).

4. Create trace templates that contain the settings specified to capture the events.

Trace templates include specifications about the events themselves, event data, and filters that are used to capture data. These templates can be used to monitor a specific set of events later without redefining the events, event data, and filters. For example, if you want to frequently monitor the number of deadlocks, and the users involved in those deadlocks, you can create a template defining those events, event data, and event filters; save the template; and reapply the filter the next time that you want to monitor deadlocks. SQL Server Profiler uses trace templates for this purpose. For more information, see [Set Trace Definition Defaults \(SQL Server Profiler\)](#) and [Create a Trace Template \(SQL Server Profiler\)](#).

5. Analyze captured event data.

To be analyzed, the captured event data is loaded into the application that captured the data. For example, a captured trace from SQL Server Profiler can be reloaded into SQL Server Profiler for viewing and analysis. For more information, see [View and Analyze Traces with SQL Server Profiler](#).





Analyzing event data involves determining what is occurring and why. This information lets you make changes that can improve performance, such as adding more memory, changing indexes, correcting coding problems with Transact-SQL statements or stored procedures, and so on, depending on the type of analysis performed. For example, you can use the Database Engine Tuning Advisor to analyze a captured trace from SQL Server Profiler and make index recommendations based on the results.

6. Replay captured event data.

Event replay lets you establish a test copy of the database environment from which the data was captured, and then repeat the captured events as they occurred originally on the real system. This capability is only available with the Distributed Replay Utility or SQL Server Profiler. You can replay the events at the same speed as they originally occurred, as fast as possible (to stress the system), or more likely, one step at a time (to analyze the system after each event has occurred). By analyzing the exact events in a test environment, you can prevent harm to the production system. For more information, see [Replay Traces](#).

Performance Monitoring and Tuning Tools

1/17/2018 • 4 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

Microsoft SQL Server provides a comprehensive set of tools for monitoring events in SQL Server and for tuning the physical database design. The choice of tool depends on the type of monitoring or tuning to be done and the particular events to be monitored.

Following are the SQL Server monitoring and tuning tools:

TOOL	DESCRIPTION
sp_trace_setfilter (Transact-SQL)	SQL Server Profiler tracks engine process events, such as the start of a batch or a transaction, enabling you to monitor server and database activity (for example, deadlocks, fatal errors, or login activity). You can capture SQL Server Profiler data to a SQL Server table or a file for later analysis, and you can also replay the events captured on SQL Server step by step, to see exactly what happened.
SQL Server Distributed Replay	Microsoft SQL Server Distributed Replay can use multiple computers to replay trace data, simulating a mission-critical workload.
Monitor Resource Usage (System Monitor)	<p>System Monitor primarily tracks resource usage, such as the number of buffer manager page requests in use, enabling you to monitor server performance and activity using predefined objects and counters or user-defined counters to monitor events. System Monitor (Performance Monitor in Microsoft Windows NT 4.0) collects counts and rates rather than data about the events (for example, memory usage, number of active transactions, number of blocked locks, or CPU activity). You can set thresholds on specific counters to generate alerts that notify operators.</p> <p>System Monitor works on Microsoft Windows Server and Windows operating systems. It can monitor (remotely or locally) an instance of SQL Server on Windows NT 4.0 or later.</p> <p>The key difference between SQL Server Profiler and System Monitor is that SQL Server Profiler monitors Database Engine events, whereas System Monitor monitors resource usage associated with server processes.</p>
Open Activity Monitor (SQL Server Management Studio)	<p>The Activity Monitor in SQL Server Management Studio is useful for ad hoc views of current activity and graphically displays information about:</p> <ul style="list-style-type: none">Processes running on an instance of SQL Server.Blocked processes.Locks.User activity.

TOOL	DESCRIPTION
Live Query Statistics	Displays real-time statistics about query execution steps. Because this data is available while the query is executing, these execution statistics are extremely useful for debugging query performance issues.
SQL Trace	<p>Transact-SQL stored procedures that create, filter, and define tracing:</p> <p>sp_trace_create (Transact-SQL)</p> <p>sp_trace_generateevent (Transact-SQL)</p> <p>sp_trace_setevent (Transact-SQL)</p> <p>sp_trace_setfilter (Transact-SQL)</p> <p>sp_trace_setstatus (Transact-SQL)</p>
Error Logs	The Windows application event log provides an overall picture of events occurring on the Windows Server and Windows operating systems as a whole, as well as events in SQL Server, SQL Server Agent, and full-text search. It contains information about events in SQL Server that is not available elsewhere. You can use the information in the error log to troubleshoot SQL Server-related problems.
System Stored Procedures (Transact-SQL)	<p>The following SQL Server system stored procedures provide a powerful alternative for many monitoring tasks:</p> <p>sp_who (Transact-SQL): Reports snapshot information about current SQL Server users and processes, including the currently executing statement and whether the statement is blocked.</p> <p>sp_lock (Transact-SQL): Reports snapshot information about locks, including the object ID, index ID, type of lock, and type or resource to which the lock applies.</p> <p>sp_spaceused (Transact-SQL): Displays an estimate of the current amount of disk space used by a table (or a whole database).</p> <p>sp_monitor (Transact-SQL): Displays statistics, including CPU usage, I/O usage, and the amount of time idle since sp_monitor was last executed.</p>
DBCC (Transact-SQL)	DBCC (Database Console Command) statements enable you to check performance statistics and the logical and physical consistency of a database.
Built-in Functions (Transact-SQL)	<p>Built-in functions display snapshot statistics about SQL Server activity since the server was started; these statistics are stored in predefined SQL Server counters. For example, @@CPU_BUSY contains the amount of time the CPU has been executing SQL Server code; @@CONNECTIONS contains the number of SQL Server connections or attempted connections; and @@PACKET_ERRORS contains the number of network packets occurring on SQL Server connections.</p>

TOOL	DESCRIPTION
Trace Flags (Transact-SQL)	Trace flags display information about a specific activity within the server and are used to diagnose problems or performance issues (for example, deadlock chains).
Database Engine Tuning Advisor	Database Engine Tuning Advisor analyzes the performance effects of Transact-SQL statements executed against databases you want to tune. Database Engine Tuning Advisor provides recommendations to add, remove, or modify indexes, indexed views, and partitioning.

Choosing a Monitoring Tool

The choice of a monitoring tool depends on the event or activity to be monitored.

EVENT OR ACTIVITY	SQL SERVER PROFILER	DISTRIBUTED REPLAY	SYSTEM MONITOR	ACTIVITY MONITOR	TRANSACTION-SQL	ERROR LOGS
Trend analysis	Yes		Yes			
Replaying captured events	Yes (From a single computer)	Yes (From multiple computers)				
Ad hoc monitoring	Yes			Yes	Yes	Yes
Generating alerts			Yes			
Graphical interface	Yes		Yes	Yes		Yes
Using within custom application	Yes*				Yes	

*Using SQL Server Profiler system stored procedures.

Windows Monitoring Tools





Windows operating systems and Windows Server 2003 also provide these monitoring tools.

TOOL	DESCRIPTION
Task Manager	Shows a synopsis of the processes and applications running on the system.
Network Monitor Agent	Monitors network traffic.

For more information about Windows operating systems or Windows Server tools, see the Windows documentation.

Establish a Performance Baseline

1/17/2018 • 1 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

To determine whether your SQL Server system is performing optimally, take performance measurements at regular intervals over time, even when no problems occur, to establish a server performance baseline. Compare each new set of measurements with those taken earlier.

The following areas affect the performance of SQL Server:

- System resources (hardware)
- Network architecture
- The operating system
- Database applications
- Client applications

At a minimum, use baseline measurements to determine:

- Peak and off-peak hours of operation.
- Production-query or batch-command response times.
- Database backup and restore completion times.





After you establish a server performance baseline, compare the baseline statistics to current server performance. Numbers far above or far below your baseline are candidates for further investigation. They may indicate areas in need of tuning or reconfiguration. For example, if the amount of time to execute a set of queries increases, examine the queries to determine if they can be rewritten, or if column statistics or new indexes must be added.

See Also

[sp_configure \(Transact-SQL\)](#)

Isolate Performance Problems

1/17/2018 • 1 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

It is often more effective to use several Microsoft SQL Server or Microsoft Windows tools together to isolate database performance problems than to use one tool at a time. For example, the graphical Execution Plan feature, also called Showplan, helps you quickly recognize deadlocks in a single query. However, you can recognize some other performance problems more easily if you use the monitoring features of SQL Server and Windows together.

SQL Server Profiler can be used to monitor and troubleshoot Transact-SQL and application-related problems. System Monitor can be used to monitor hardware and other system-related problems.

You can monitor the following areas to troubleshoot problems:

- SQL Server stored procedures or batches of Transact-SQL statements submitted by user applications.
- User activity, such as blocking locks or deadlocks.
- Hardware activity, such as disk usage.

Problems can include:

- Application development errors involving incorrectly written Transact-SQL statements.
- Hardware errors, such as disk- or network-related errors.
- Excessive blocking due to an incorrectly designed database.

Tools for Common Performance Problems

Equally important is careful selection of the performance problem that you want each tool to monitor or tune. The tool and the utility depend on the type of performance problem you want to resolve.





The following topics describe a variety of monitoring and tuning tools and the problems they uncover.

[Identify Bottlenecks](#)

[Monitor Memory Usage](#)

Identify Bottlenecks

1/17/2018 • 1 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

Simultaneous access to shared resources causes bottlenecks. In general, bottlenecks are present in every software system and are inevitable. However, excessive demands on shared resources cause poor response time and must be identified and tuned.

Causes of bottlenecks include:

- Insufficient resources, requiring additional or upgraded components.
- Resources of the same type among which workloads are not distributed evenly; for example, one disk is being monopolized.
- Malfunctioning resources.
- Incorrectly configured resources.

Analyzing Bottlenecks

Excessive durations for various events are indicators of bottlenecks that can be tuned.

For example:

- Some other component may prevent the load from reaching this component thereby increasing the time to complete the load.
- Client requests may take longer due to network congestion.

Following are five key areas to monitor when tracking server performance to identify bottlenecks.

POSSIBLE BOTTLENECK AREA	EFFECTS ON THE SERVER
Memory usage	Insufficient memory allocated or available to Microsoft SQL Server degrades performance. Data must be read from the disk rather than directly from the data cache. Microsoft Windows operating systems perform excessive paging by swapping data to and from the disk as the pages are needed.
CPU utilization	A chronically high CPU utilization rate may indicate that Transact-SQL queries need to be tuned or that a CPU upgrade is needed.
Disk input/output (I/O)	Transact-SQL queries can be tuned to reduce unnecessary I/O; for example, by employing indexes.
User connections	Too many users may be accessing the server simultaneously causing performance degradation.
Blocking locks	Incorrectly designed applications can cause locks and hamper concurrency, thus causing longer response times and lower transaction throughput rates.

See Also

[Monitor CPU Usage](#)

[Monitor Disk Usage](#)





[Monitor Memory Usage](#)

[SQL Server, General Statistics Object](#)

[SQL Server, Locks Object](#)

Server Performance and Activity Monitoring

1/17/2018 • 3 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

The goal of monitoring databases is to assess how a server is performing. Effective monitoring involves taking periodic snapshots of current performance to isolate processes that are causing problems, and gathering data continuously over time to track performance trends. Microsoft SQL Server and the Microsoft Windows operating system provide utilities that let you view the current condition of the database and to track performance as conditions change.

The following section contains topics that describe how to use SQL Server and Windows performance and activity monitoring tools. It contains the following topics:

In This Section

To perform monitoring tasks with Windows tools

- [Start System Monitor \(Windows\)](#)
- [View the Windows Application Log \(Windows\)](#)

To create SQL Server database alerts with Windows tools

- [Set Up a SQL Server Database Alert \(Windows\)](#)

To perform monitoring tasks with Extended Events

- [Extended Events](#)
- [Quick Start: Extended events in SQL Server](#)

To perform monitoring tasks with SQL Server Management Studio

- [View the SQL Server Error Log \(SQL Server Management Studio\)](#)
- [Open Activity Monitor \(SQL Server Management Studio\)](#)

To perform monitoring tasks with SQL Trace by using Transact-SQL stored procedures

- [Create a Trace \(Transact-SQL\)](#)
- [Set a Trace Filter \(Transact-SQL\)](#)
- [Modify an Existing Trace \(Transact-SQL\)](#)
- [View a Saved Trace \(Transact-SQL\)](#)
- [View Filter Information \(Transact-SQL\)](#)
- [Delete a Trace \(Transact-SQL\)](#)

To create and modify traces by using SQL Server Profiler

- [Create a Trace \(SQL Server Profiler\)](#)
- [Set Global Trace Options \(SQL Server Profiler\)](#)

- [Specify Events and Data Columns for a Trace File \(SQL Server Profiler\)](#)
- [Create a Transact-SQL Script for Running a Trace \(SQL Server Profiler\)](#)
- [Save Trace Results to a File \(SQL Server Profiler\)](#)
- [Set a Maximum File Size for a Trace File \(SQL Server Profiler\)](#)
- [Save Trace Results to a Table \(SQL Server Profiler\)](#)
- [Set a Maximum Table Size for a Trace Table \(SQL Server Profiler\)](#)
- [Filter Events in a Trace \(SQL Server Profiler\)](#)
- [View Filter Information \(SQL Server Profiler\)](#)
- [Modify a Filter \(SQL Server Profiler\)](#)
- [Filter Events Based on the Event Start Time \(SQL Server Profiler\)](#)
- [Filter Events Based on the Event End Time \(SQL Server Profiler\)](#)
- [Filter Server Process IDs \(SPIDs\) in a Trace \(SQL Server Profiler\)](#)
- [Organize Columns Displayed in a Trace \(SQL Server Profiler\)](#)

To start, pause, and stop traces by using SQL Server Profiler

- [Start a Trace Automatically after Connecting to a Server \(SQL Server Profiler\)](#)
- [Pause a Trace \(SQL Server Profiler\)](#)
- [Stop a Trace \(SQL Server Profiler\)](#)
- [Run a Trace After It Has Been Paused or Stopped \(SQL Server Profiler\)](#)

To open traces and configure how traces are displayed by using SQL Server Profiler

- [Open a Trace File \(SQL Server Profiler\)](#)
- [Open a Trace Table \(SQL Server Profiler\)](#)
- [Clear a Trace Window \(SQL Server Profiler\)](#)
- [Close a Trace Window \(SQL Server Profiler\)](#)
- [Set Trace Definition Defaults \(SQL Server Profiler\)](#)
- [Set Trace Display Defaults \(SQL Server Profiler\)](#)

To replay traces by using SQL Server Profiler

- [Replay a Trace File \(SQL Server Profiler\)](#)
- [Replay a Trace Table \(SQL Server Profiler\)](#)
- [Replay a Single Event at a Time \(SQL Server Profiler\)](#)
- [Replay to a Breakpoint \(SQL Server Profiler\)](#)
- [Replay to a Cursor \(SQL Server Profiler\)](#)
- [Replay a Transact-SQL Script \(SQL Server Profiler\)](#)

To create, modify, and use trace templates by using SQL Server Profiler





- [Create a Trace Template \(SQL Server Profiler\)](#)
- [Modify a Trace Template \(SQL Server Profiler\)](#)
- [Derive a Template from a Running Trace \(SQL Server Profiler\)](#)
- [Derive a Template from a Trace File or Trace Table \(SQL Server Profiler\)](#)
- [Export a Trace Template \(SQL Server Profiler\)](#)
- [Import a Trace Template \(SQL Server Profiler\)](#)

To use SQL Server Profiler traces to collect and monitor server performance

- [Find a Value or Data Column While Tracing \(SQL Server Profiler\)](#)
- [Save Deadlock Graphs \(SQL Server Profiler\)](#)
- [Save Showplan XML Events Separately \(SQL Server Profiler\)](#)
- [Save Showplan XML Statistics Profile Events Separately \(SQL Server Profiler\)](#)
- [Extract a Script from a Trace \(SQL Server Profiler\)](#)
- [Correlate a Trace with Windows Performance Log Data \(SQL Server Profiler\)](#)

Start System Monitor (Windows)

1/17/2018 • 1 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

Use System Monitor to monitor the utilization of system resources. Collect and view real-time performance data in the form of counters. You can use counters for server resources, such as processor and memory use. You also can use them for many Microsoft SQL Server resources, such as locks and transactions.

Start System Monitor in Windows





On the **Start** menu, point to **Run**, type **perfmon** in the **Run** dialog box, and then select **OK**.

See also

[Run System Monitor](#)

Set up a SQL Server database alert (Windows)

1/17/2018 • 1 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

You can use System Monitor to create an alert that is raised when a System Monitor counter reaches a threshold value. In response to the alert, System Monitor can launch an application, such as a custom application written to handle the alert condition. For example, you can create an alert that is raised when the number of deadlocks exceeds a specific value.

Alerts also can be defined by using Microsoft SQL Server Management Studio and SQL Server Agent. For more information, see [Alerts](#).

Set up a SQL Server database alert

1. On the navigation tree of the **Performance** window, expand **Performance Logs and Alerts**.
2. Right-click **Alerts**, and then select **New Alert Settings**.
3. In the **New Alert Settings** dialog box, type a name for the new alert, and then select **OK**.
4. On the **General** tab of the dialog box for the new alert, add a **Comment**. Select **Add** to add a counter to the alert.

All alerts must have at least one counter.

5. In the **Add Counters** dialog box, select a SQL Server object from the **Performance Object** list. Select a counter from the **Select counters from** list.
6. To add the counter to the alert, select **Add**. You can continue to add counters, or you can select **Close** to return to the dialog box for the new alert.
7. In the new alert dialog box, select either **Over** or **Under** in the **Alert when the value is** list. Then enter a threshold value in **Limit**.

The alert is generated when the value for the counter is more than or less than the threshold value (depending on whether you selected **Over** or **Under**).





8. In the **Sample data every** boxes, set the sampling frequency.
9. On the **Action** tab, set actions to occur every time the alert is triggered.
10. On the **Schedule** tab, set the start and stop schedule for the alert scan.

See also

[Create a SQL Server database alert](#)

View the Windows application log (Windows 10)

1/17/2018 • 1 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

When SQL Server is configured to use the Windows application log, each SQL Server session writes new events to that log. Unlike the SQL Server error log, a new application log is not created each time you start an instance of SQL Server.

View the Windows application log





1. On the **Search bar**, type **Event Viewer**, and then select the **Event Viewer** desktop app.
2. In **Event Viewer**, open the **Applications and Services Logs**.
3. SQL Server events are identified by the entry **MSSQLSERVER** (named instances are identified with **MSSQL\$<instance_name>**) in the **Source** column. SQL Server Agent events are identified by the entry **SQLSERVERAGENT** (for named instances of SQL Server, SQL Server Agent events are identified with **SQLAgent\$<instance_name>**). Microsoft Search service events are identified by the entry **Microsoft Search**.
4. To view the log of a different computer, right-click **Event Viewer (local)**. Select **Connect to another computer**, and fill in the fields to complete the **Select Computer** dialog box.
5. Optionally, to display only SQL Server events, on the **View** menu, select **Filter**. In the **Event source** list, select **MSSQLSERVER**. To view only SQL Server Agent events, instead select **SQLSERVERAGENT** in the **Event source** list.
6. To view more information about an event, double-click the event.

See also

[View the SQL Server error log \(SQL Server Management Studio\)](#)

View the SQL Server error log (SQL Server Management Studio)

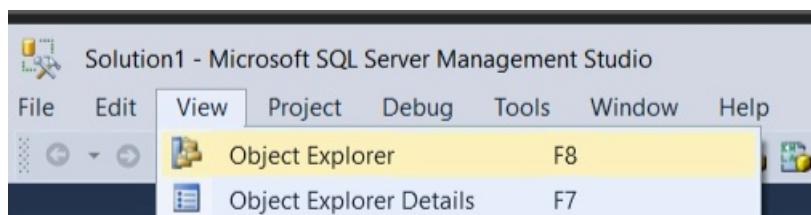
1/17/2018 • 1 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

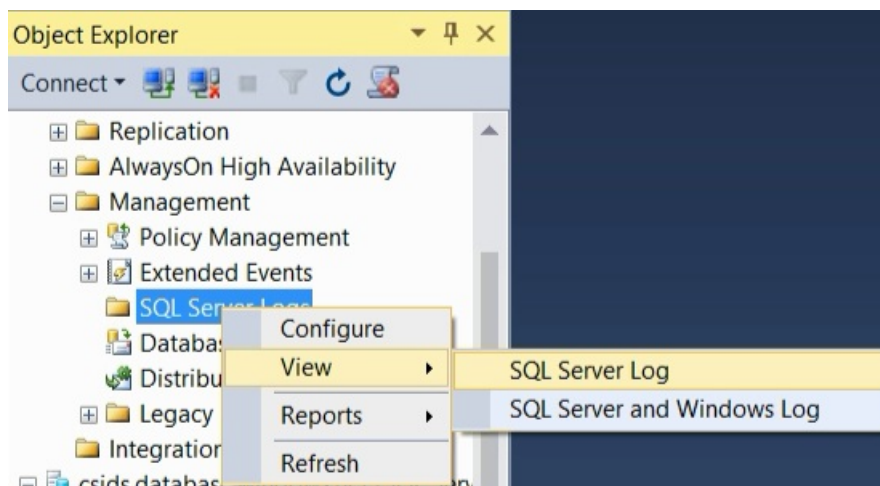
The SQL Server error log contains user-defined events and certain system events you can use for troubleshooting.

View the logs

1. In SQL Server Management Studio, select **Object Explorer**. To open **Object Explorer**, select F8. Or on the top menu, select **View**, and then select **Object Explorer**:



2. In **Object Explorer**, connect to an instance of SQL Server, and then expand that instance.
3. Find and expand the **Management** section (assuming you have permissions to see it).
4. Right-click **SQL Server Logs**, select **View**, and then choose **SQL Server Log**.







5. The **Log File Viewer** appears (it might take a moment) with a list of logs for you to view.

See also

For more information, see [MSSQLTips.com's](#) helpful post [Identify location of the SQL Server Error Log file](#).

Save deadlock graphs (SQL Server Profiler)

1/17/2018 • 1 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

This topic describes how to save a deadlock graph by using SQL Server Profiler. Deadlock graphs are saved as XML files.

Save deadlock graph events separately

1. On the **File** menu, select **New Trace**, and then connect to an instance of SQL Server.

The **Trace Properties** dialog box appears.

NOTE

If you select **Start tracing immediately after making connection**, the **Trace Properties** dialog box fails to appear, and the trace begins instead. To turn off this setting, on the **Tools** menu, select **Options**, and clear the **Start tracing immediately after making connection** check box.

2. In the **Trace Properties** dialog box, type a name for the trace in the **Trace name** box.
3. In the **Use the template** list, select a trace template on which to base the trace. If you don't want to use a template, select **Blank**.
4. Do one of the following:

- To capture the trace to a file, select the **Save to file** check box. Specify a value for **Set maximum file size**.

Optionally, select the **Enable file rollover** and **Server processes trace data** check boxes.

- To capture the trace to a database table, select the **Save to table** check box.

Optionally, select **Set maximum rows**, and specify a value.

5. Optionally, select the **Enable trace stop time** check box, and specify a stop date and time.

6. Select the **Events Selection** tab.

7. In the **Events** data column, expand the **Locks** event category, and then select the **Deadlock graph** check box. If the **Locks** event category isn't available, select the **Show all events** check box to display it.

The **Events Extraction Settings** tab is added to the **Trace Properties** dialog box.

8. On the **Events Extraction Settings** tab, select **Save Deadlock XML Events Separately**.

9. In the **Save As** dialog box, enter the name of the file where you want to store the deadlock graph events.

10. Select **All Deadlock XML batches in a single file** to save all deadlock graph events in a single XML file. Or select **Each Deadlock XML batch in a distinct file** to create a new XML file for each deadlock graph.





After you save the deadlock file, you can open the file in SQL Server Management Studio. For more information, see [Open, view, and print a deadlock file \(SQL Server Management Studio\)](#).

See also

[Analyze deadlocks with SQL Server Profiler](#)

Open, view, and print a deadlock file (SQL Server Management Studio)

1/17/2018 • 1 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

When SQL Server Profiler generates a deadlock, you can capture and save the deadlock information to a file. After you've saved the deadlock file, you can open it in SQL Server Management Studio to view it or print it.

Open and view a deadlock file

1. On the **File** menu in SQL Server Management Studio, point to **Open**, and then select **File**.
2. In the **Open File** dialog box, select the .xdl file type in the **Files of type** box. You now have a filtered list of only deadlock files.

Print a deadlock file





1. On the **File** menu in SQL Server Management Studio, point to **Open**, and then select **File**.
2. In the **Open File** dialog box, select the .xdl file type in the **Files of type** box. You now have a filtered list of only deadlock files.
3. Select the deadlock file you want to print, and select **Open**.
4. On the **File** menu, select **Print**.

See also

[Save deadlock graphs \(SQL Server Profiler\)](#)

Save Showplan XML events separately (SQL Server Profiler)

1/17/2018 • 2 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

This topic describes how to save **Showplan XML** events that are captured in traces into separate .SQLPlan files by using SQL Server Profiler. You can open the **Showplan XML** event files in SQL Server Management Studio so that you can view the graphical execution plan for each event.

Save Showplan XML events separately

1. On the **File** menu, select **New Trace**, and then connect to an instance of SQL Server.

The **Trace Properties** dialog box appears.

NOTE

If you select **Start tracing immediately after making connection**, the **Trace Properties** dialog box fails to appear, and the trace begins instead. To turn off this setting, on the **Tools** menu, select **Options**, and clear the **Start tracing immediately after making connection** check box.

2. In the **Trace Properties** dialog box, type a name for the trace in the **Trace name** box.
3. In the **Use the template** list, select a trace template on which to base the trace. If you don't want to use a template, select **Blank**.
4. Do one of the following:
 - To capture the trace to a file, select the **Save to file** check box. Specify a value for **Set maximum file size**.
Optionally, select the **Enable file rollover** and **Server processes trace data** check boxes.
 - To capture the trace to a database table, select the **Save to table** check box.
Optionally, select **Set maximum rows**, and specify a value.
5. Optionally, select the **Enable trace stop time** check box, and specify a stop date and time.
6. Select the **Events Selection** tab.
7. In the **Events** data column, expand the **Performance** event category, and then select the **Showplan XML** check box. If the **Performance** event category is not available, select **Show all events** to display it.

The **Events Extraction Settings** tab is added to the **Trace Properties** dialog box.
8. On the **Events Extraction Settings** tab, select **Save XML Showplan events separately**.
9. In the **Save As** dialog box, enter the name of the file in which to store the **Showplan XML** events.
10. Select **All XML Showplan batches in a single file** to save all **Showplan XML** events in a single XML file. Or select **Each XML Showplan batch in a distinct file** to create a new XML file for each **Showplan XML** event.





11. To view the **Showplan XML** event file in SQL Server Management Studio, on the **File** menu, point to **Open**, and select **File**. Browse to the directory where you saved the **Showplan XML** event file or files to select one and open it. **Showplan XML** event files have a .SQLPlan file extension.

See also

[Analyze queries with Showplan results in SQL Server Profiler](#)

Save Showplan XML Statistics Profile events separately (SQL Server Profiler)

1/17/2018 • 2 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

This topic describes how to save **Showplan XML Statistics Profile** events that are captured in traces into separate .SQLPlan files by using SQL Server Profiler. You can open the **Showplan XML Statistics Profile** event files in SQL Server Management Studio so that you can view the graphical execution plan for each event.

Save Showplan XML Statistics Profile events separately

1. On the **File** menu, select **New Trace**, and then connect to an instance of SQL Server.

The **Trace Properties** dialog box appears.

NOTE

If you select **Start tracing immediately after making connection**, the **Trace Properties** dialog box doesn't appear, and the trace begins instead. To turn off this setting, on the **Tools** menu, select **Options**, and clear the **Start tracing immediately after making connection** check box.

2. In the **Trace Properties** dialog box, type a name for the trace in the **Trace name** box.
3. In the **Use the template** list, select a trace template on which to base the trace. If you don't want to use a template, select **Blank**.
4. Do one of the following:
 - To capture the trace to a file, select the **Save to file** check box. Specify a value for **Set maximum file size**.
Optionally, select the **Enable file rollover** and **Server processes trace data** check boxes.
 - To capture the trace to a database table, select the **Save to table** check box.
Optionally, select **Set maximum rows**, and specify a value.
5. Optionally, select the **Enable trace stop time** check box, and specify a stop date and time.
6. Select the **Events Selection** tab.
7. In the **Events** data column, expand the **Performance** event category, and then select the **Showplan XML Statistics Profile** check box. If the **Performance** event category isn't available, check **Show all events** to display it.

The **Events Extraction Settings** tab is added to the **Trace Properties** dialog box.

8. On the **Events Extraction Settings** tab, select **Save XML Showplan events separately**.
9. In the **Save As** dialog box, enter the file name to store the **Showplan XML Statistics Profile** events.
10. Select **All batches in a single file** to save all **Showplan XML Statistics Profile** events in a single XML file. Or select **Each XML Showplan batch in a distinct file** to create a new XML file for each **Showplan XML**

Statistics Profile event.





11. To view the **Showplan XML Statistics Profile** event file in SQL Server Management Studio, on the **File** menu, point to **Open**, and select **File**. Browse to the directory where you saved the **Showplan XML Statistics Profile** event file or files to select one and open it. **Showplan XML Statistics Profile** event files have a .SQLPlan file extension.

See also

[Analyze queries with Showplan results in SQL Server Profiler](#)

Display and Save Execution Plans

1/17/2018 • 1 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

This section explains how to display execution plans and how to save execution plans to a file in XML format by using Microsoft SQL Server Management Studio.

Execution plans graphically display the data retrieval methods chosen by the SQL Server Query Optimizer. Execution plans represent the execution cost of specific statements and queries in SQL Server using icons rather than the tabular representation produced by the [SET SHOWPLAN_ALL](#) or [SET SHOWPLAN_TEXT](#) statements. This graphical approach is very useful for understanding the performance characteristics of a query.

While the SQL Server Query Optimizer produces only one execution plan, there is the concept of **estimated** execution plan and **actual** execution plan.

- An [estimated execution plan](#) returns the execution plan as produced by the Query Optimizer at compile-time. Producing the estimated execution plan does not actually execute the query or batch, and therefore does not contain any runtime information, such as actual resource usage metrics or runtime warnings.
- An [actual execution plan](#) returns the execution plan as produced by the Query Optimizer, and after queries or batches finish execution. This includes runtime information about resource usage metrics and any runtime warnings.





For more information, see [Query Processing Architecture Guide](#).

In This Section

- [Display the Estimated Execution Plan](#)
- [Display an Actual Execution Plan](#)
- [Save an Execution Plan in XML Format](#)

Display the Estimated Execution Plan

1/17/2018 • 2 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

This topic describes how to generate graphical estimated execution plans by using SQL Server Management Studio. When estimated execution plans are generated, the Transact-SQL queries or batches do not execute. Because of this, an estimated execution plan does not contain any runtime information, such as actual resource usage metrics or runtime warnings. Instead, the execution plan that is generated displays the query execution plan that SQL Server Database Engine would most probably use if the queries were actually executed, and displays the estimated rows flowing through the several operators in the plan.

To use this feature, users must have the appropriate permissions to execute the Transact-SQL query for which a graphical execution plan is being generated, and they must be granted the SHOWPLAN permission for all databases referenced by the query.

To display the estimated execution plan for a query





1. On the toolbar, click **Database Engine Query**. You can also open an existing query and display the estimated execution plan by clicking the **Open File** toolbar button and locating the existing query.
2. Enter the query for which you would like to display the estimated execution plan.
3. On the **Query** menu, click **Display Estimated Execution Plan** or click the **Display Estimated Execution Plan** toolbar button. The estimated execution plan is displayed on the **Execution Plan** tab in the results pane. To view additional information, pause the mouse over the logical and physical operator icons and view the description and properties of the operator in the displayed ToolTip. Alternatively, you can view operator properties in the Properties window. If Properties is not visible, right-click an operator and click **Properties**. Select an operator to view its properties.
4. To alter the display of the execution plan, right-click the execution plan and select **Zoom In**, **Zoom Out**, **Custom Zoom**, or **Zoom to Fit**. **Zoom In** and **Zoom Out** allow you to magnify or reduce the execution plan by fixed amounts. **Custom Zoom** allows you to define your own display magnification, such as zooming at 80 percent. **Zoom to Fit** magnifies the execution plan to fit the result pane. Alternatively, use a combination of the CTRL key and your mouse wheel to activate **dynamic zoom**.

NOTE

Alternatively, use [SET SHOWPLAN_XML](#) to return execution plan information for each statement without executing it. If used in SQL Server Management Studio, the *Results* tab will have a link to open the execution plan in graphical format.

Display an Actual Execution Plan

1/17/2018 • 2 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

This topic describes how to generate actual graphical execution plans by using SQL Server Management Studio. Actual execution plans are generated after the Transact-SQL queries or batches execute. Because of this, an actual execution plan contains runtime information, such as actual resource usage metrics and runtime warnings (if any). The execution plan that is generated displays the actual query execution plan that the SQL Server Database Engine used to execute the queries.

To use this feature, users must have the appropriate permissions to execute the Transact-SQL queries for which a graphical execution plan is being generated, and they must be granted the SHOWPLAN permission for all databases referenced by the query.

To include an execution plan for a query during execution

1. On the SQL Server Management Studio toolbar, click **Database Engine Query**. You can also open an existing query and display the estimated execution plan by clicking the **Open File** toolbar button and locating the existing query.
2. Enter the query for which you would like to display the actual execution plan.
3. On the **Query** menu, click **Include Actual Execution Plan** or click the **Include Actual Execution Plan** toolbar button.
4. Execute the query by clicking the **Execute** toolbar button. The plan used by the query optimizer is displayed on the **Execution Plan** tab in the results pane. Pause the mouse over the logical and physical operators to view the description and properties of the operators in the displayed ToolTip.

Alternatively, you can view operator properties in the Properties window. If Properties is not visible, right-click an operator and select **Properties**. Select an operator to view its properties.





5. You can alter the display of the execution plan by right-clicking the execution plan and selecting **Zoom In**, **Zoom Out**, **Custom Zoom**, or **Zoom to Fit**. **Zoom In** and **Zoom Out** allow you to zoom in or out on the execution plan, while **Custom Zoom** allows you to define your own zoom, such as zooming at 80 percent. **Zoom to Fit** magnifies the execution plan to fit the result pane. Alternatively, use a combination of the CTRL key and your mouse wheel to activate **dynamic zoom**.

NOTE

Alternatively, use [SET STATISTICS XML](#) to return execution plan information for each statement after executing it. If used in SQL Server Management Studio, the *Results* tab will have a link to open the execution plan in graphical format.

Save an Execution Plan in XML Format

1/17/2018 • 2 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

Use SQL Server Management Studio to save execution plans as an XML file, and to open them for viewing.

To use the execution plan feature in Management Studio, or to use the XML Showplan SET options, users must have the appropriate permissions to execute the Transact-SQL query for which an execution plan is being generated, and they must be granted the SHOWPLAN permission for all databases referenced by the query.

To save a query plan by using the XML Showplan SET options

1. In SQL Server Management Studio open a query editor and connect to Database Engine.
2. Turn [SHOWPLAN_XML](#) on with the following statement:

```
SET SHOWPLAN_XML ON;  
GO
```

To turn [STATISTICS XML](#) on, use the following statement:

```
SET STATISTICS XML ON;  
GO
```

NOTE

SHOWPLAN_XML generates compile-time query execution plan information for a query, but does not execute the query. This is also known as the **estimated** execution plan. STATISTICS XML generates runtime query execution plan information for a query, and executes the query. This is also known as the **actual** execution plan.

3. Execute a query. Example:

```
USE AdventureWorks2012;  
GO  
SET SHOWPLAN_XML ON;  
GO  
-- Execute a query.  
SELECT BusinessEntityID  
FROM HumanResources.Employee  
WHERE NationalIDNumber = '509647174';  
GO  
SET SHOWPLAN_XML OFF;
```

4. In the **Results** pane, right-click the **Microsoft SQL Server XML Showplan** that contains the query plan, and then click **Save Results As**.
5. In the **Save <Grid or Text> Results** dialog box, in the **Save as type** box, click **All files (*.*)**.
6. In the **File name** box provide a name, in the format <name>.sqlplan, and then click **Save**.

To save an execution plan by using SQL Server Management Studio options

1. Generate either an estimated execution plan or an actual execution plan by using Management Studio. For more information, see [Display the Estimated Execution Plan](#) and [Display an Actual Execution Plan](#).
2. In the **Execution plan** tab of the results pane, right-click the graphical execution plan, and choose **Save Execution Plan As**.

As an alternative, you can also choose **Save Execution Plan As** on the **File** menu.

3. In the **Save As** dialog box, make sure that the **Save as type** is set to **Execution Plan Files (*.sqlplan)**.
4. In the **File name** box provide a name, in the format <name>.sqlplan, and then click **Save**.

To open a saved XML query plan in SQL Server Management Studio

1. In SQL Server Management Studio, on the **File** menu, choose **Open**, and then click **File**.
2. In the **Open File** dialog box, set **Files of type** to **Execution Plan Files (*.sqlplan)** to produce a filtered list of saved XML query plan files.
3. Select the XML query plan file that you want to view, and click **Open**.

As an alternative, in Windows Explorer, double-click a file with extension **.sqlplan**. The plan opens in Management Studio.





See Also

[SET SHOWPLAN_XML \(Transact-SQL\)](#)

[SET STATISTICS XML \(Transact-SQL\)](#)

Live Query Statistics

1/17/2018 • 2 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

SQL Server Management Studio provides the ability to view the live execution plan of an active query. This live query plan provides real-time insights into the query execution process as the controls flow from one query plan operator to another. The live query plan displays the overall query progress and operator-level run-time execution statistics such as the number of rows produced, elapsed time, operator progress, etc. Because this data is available in real time without needing to wait for the query to complete, these execution statistics are extremely useful for debugging query performance issues. This feature is available beginning with SQL Server 2016 Management Studio, however it can work with SQL Server 2014.

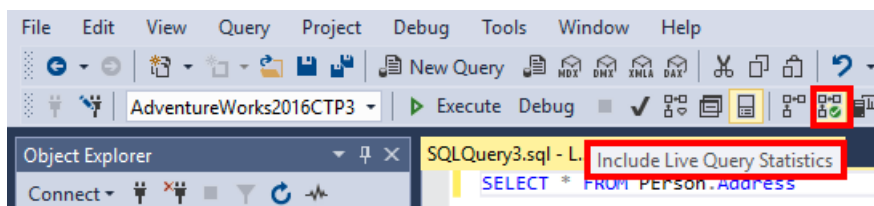
Applies to: SQL Server (SQL Server 2014 through [current version](#)).

WARNING

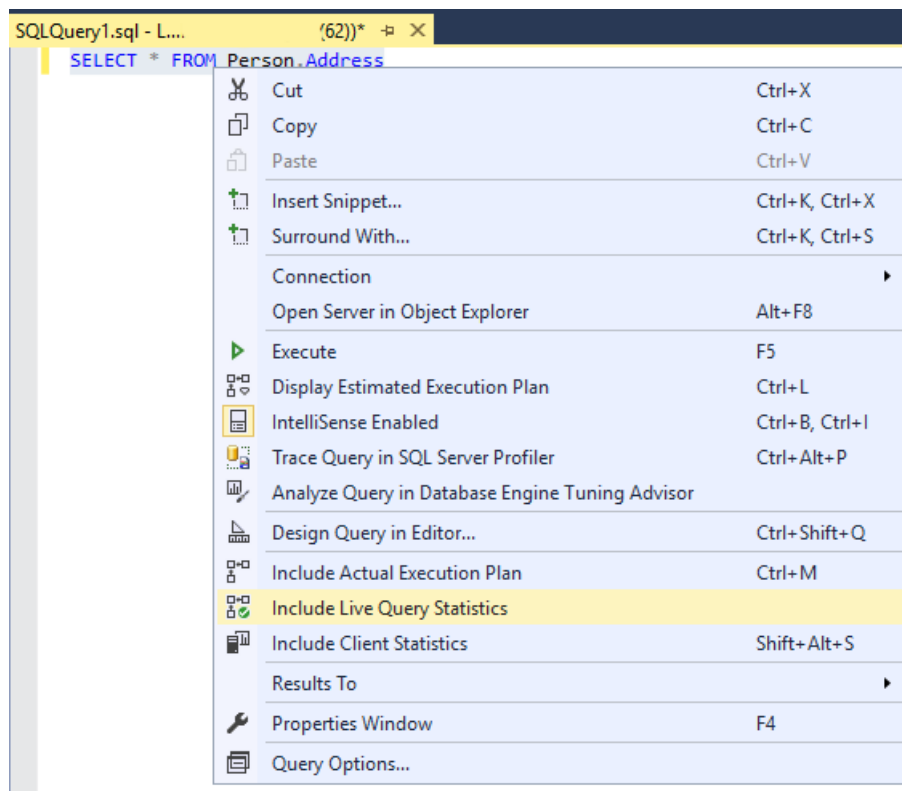
This feature is primarily intended for troubleshooting purposes. Using this feature can moderately slow the overall query performance. This feature can be used with the [Transact-SQL Debugger](#).

To view live query statistics

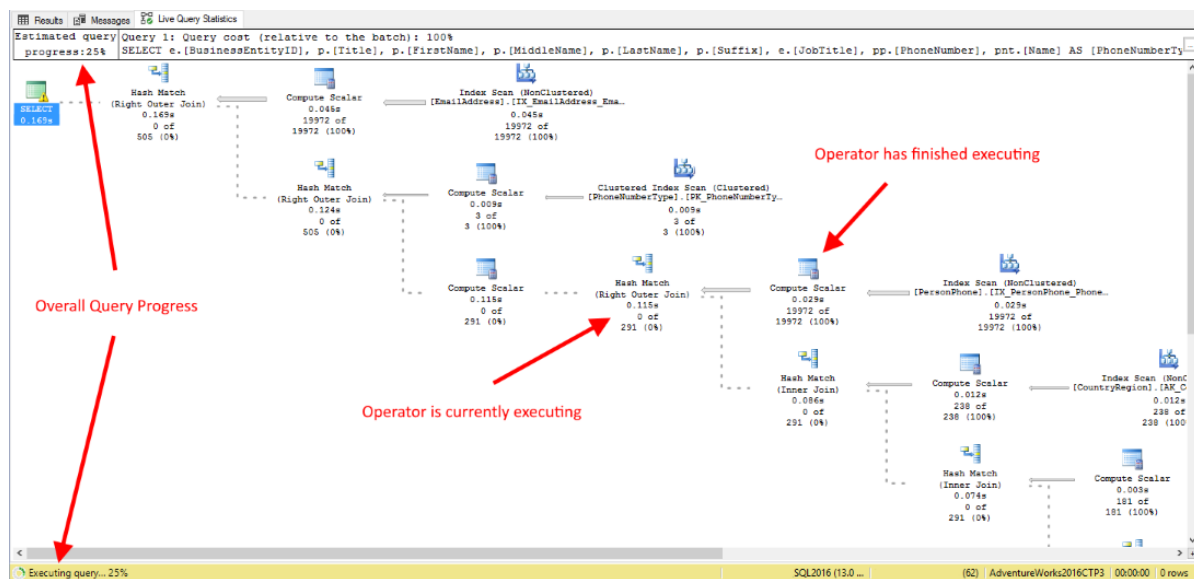
1. To view the live query execution plan, on the tools menu click the **Live Query Statistics** icon.



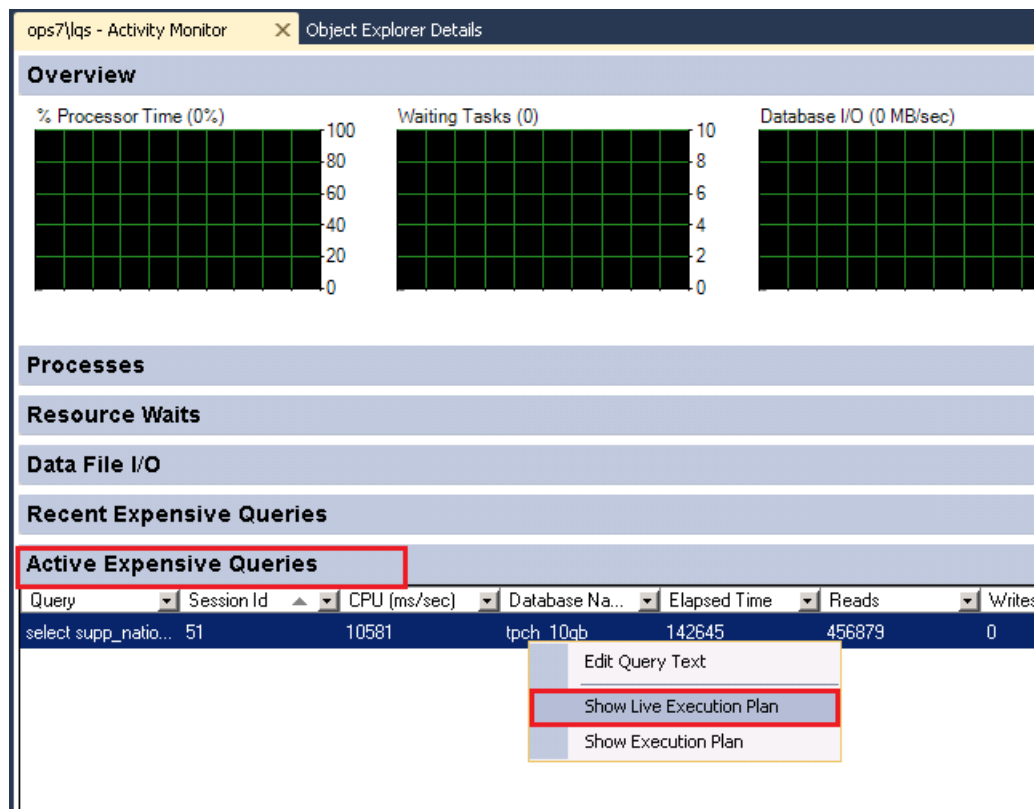
You can also view access the live query execution plan by right clicking on a selected query in Management Studio and then click **Include Live Query Statistics**.



- Now execute the query. The live query plan displays the overall query progress and the run-time execution statistics (e.g. elapsed time, progress, etc.) for the query plan operators. The query progress information and execution statistics are periodically updated while query execution is in progress. Use this information to understand the overall query execution process and to debug long running queries, queries that run indefinitely, queries that cause tempdb overflow, and timeout issues.



The live execution plan can also be accessed from the **Activity Monitor** by right-clicking on the queries in the **Active Expensive Queries** table.



Remarks

The statistics profile infrastructure must be enabled before live query statistics can capture information about the progress of queries. Specifying **Include Live Query Statistics** in Management Studio enables the statistics infrastructure for the current query session.

Until SQL Server 2016, there are two other ways to enable the statistics infrastructure which can be used to view the live query statistics from other sessions (such as from Activity Monitor):

- Execute `SET STATISTICS XML ON;` or `SET STATISTICS PROFILE ON;` in the target session.

or

- Enable the **query_post_execution_showplan** extended event. This is a server wide setting that enable live query statistics on all sessions. To enable extended events, see [Monitor System Activity Using Extended Events](#).

Starting with SQL Server 2016 SP1, SQL Server includes a lightweight version of the statistics profile infrastructure. There are two ways to enable the lightweight statistics infrastructure which can be used to view the live query statistics from other sessions (such as from Activity Monitor):

- Use global trace flag 7412.

or

- Enable the **query_thread_profile** extended event. This is a server wide setting that enable live query statistics on all sessions. To enable extended events, see [Monitor System Activity Using Extended Events](#).

NOTE

Natively compiled stored procedures are not supported.

Permissions

Requires the database level **SHOWPLAN** permission to populate the **Live Query Statistics** results page, the server level **VIEW SERVER STATE** permission to see the live statistics, and requires any permissions necessary to execute the query.

See Also

[Monitor and Tune for Performance](#)

[Performance Monitoring and Tuning Tools](#)

[Open Activity Monitor \(SQL Server Management Studio\)](#)

[Activity Monitor](#)

[Monitoring Performance By Using the Query Store](#)





[sys.dm_exec_query_statistics_xml](#)

[sys.dm_exec_query_profiles](#)

[Trace flags](#)

Monitoring performance by using the Query Store

1/17/2018 • 17 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

The SQL Server Query Store feature provides you with insight on query plan choice and performance. It simplifies performance troubleshooting by helping you quickly find performance differences caused by query plan changes. Query Store automatically captures a history of queries, plans, and runtime statistics, and retains these for your review. It separates data by time windows so you can see database usage patterns and understand when query plan changes happened on the server. You can configure query store using the [ALTER DATABASE SET](#) option.

For information about operating the Query Store in Azure SQL Database, see [Operating the Query Store in Azure SQL Database](#).

Enabling the Query Store

Query Store is not active for new databases by default.

Use the Query Store Page in SQL Server Management Studio

1. In Object Explorer, right-click a database, and then click **Properties**.

NOTE

Requires at least version 16 of Management Studio.

2. In the **Database Properties** dialog box, select the **Query Store** page.
3. In the **Operation Mode (Requested)** box, select **On**.

Use Transact-SQL Statements

1. Use the **ALTER DATABASE** statement to enable the query store. For example:

```
ALTER DATABASE AdventureWorks2012 SET QUERY_STORE = ON;
```

For more syntax options related to the query store, see [ALTER DATABASE SET Options \(Transact-SQL\)](#).

NOTE

You cannot enable the query store for the **master** or **tempdb** database.

Information in the Query Store

Execution plans for any specific query in SQL Server typically evolve over time due to a number of different reasons such as statistics changes, schema changes, creation/deletion of indexes, etc. The procedure cache (where cached query plans are stored) only stores the latest execution plan. Plans also get evicted from the plan cache due to memory pressure. As a result, query performance regressions caused by execution plan changes can be non-trivial and time consuming to resolve.

Since the query store retains multiple execution plans per query, it can enforce policies to direct the query

processor to use a specific execution plan for a query. This is referred to as plan forcing. Plan forcing in Query Store is provided by using a mechanism similar to the [USE PLAN](#) query hint, but it does not require any change in user applications. Plan forcing can resolve a query performance regression caused by a plan change in a very short period of time.

Wait stats are another source of information that helps to troubleshoot performance in SQL Server. For a long time, wait statistics were available only on instance level, which made it hard to backtrack it to the actual query. In SQL Server 2017 and Azure SQL Database we added another dimension in Query Store that tracks wait stats.

Common scenarios for using the Query Store feature are:

- Quickly find and fix a plan performance regression by forcing the previous query plan. Fix queries that have recently regressed in performance due to execution plan changes.
- Determine the number of times a query was executed in a given time window, assisting a DBA in troubleshooting performance resource problems.
- Identify top *n* queries (by execution time, memory consumption, etc.) in the past *x* hours.
- Audit the history of query plans for a given query.
- Analyze the resource (CPU, I/O, and Memory) usage patterns for a particular database.
- Identify top *n* queries that are waiting on resources.
- Understand wait nature for a particular query or plan.

The query store contains three stores:

- a **plan store** for persisting the execution plan information.
- a **runtime stats store** for persisting the execution statistics information.
- a **wait stats store** for persisting wait statistics information.

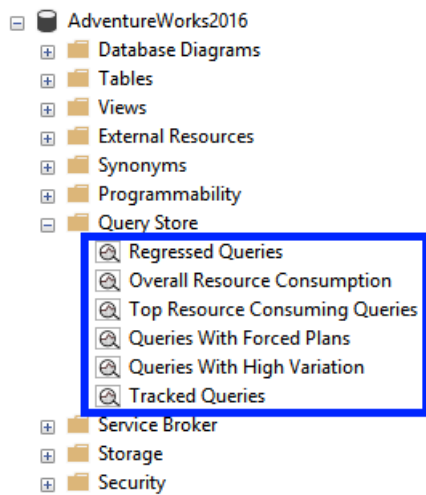
The number of unique plans that can be stored for a query in the plan store is limited by the **max_plans_per_query** configuration option. To enhance performance, the information is written to the stores asynchronously. To minimize space usage, the runtime execution statistics in the runtime stats store are aggregated over a fixed time window. The information in these stores is visible by querying the query store catalog views.

The following query returns information about queries and plans in the query store.

```
SELECT Txt.query_text_id, Txt.query_sql_text, Pl.plan_id, Qry.*
FROM sys.query_store_plan AS Pl
JOIN sys.query_store_query AS Qry
    ON Pl.query_id = Qry.query_id
JOIN sys.query_store_query_text AS Txt
    ON Qry.query_text_id = Txt.query_text_id ;
```

Use the Regressed Queries Feature

After enabling the query store, refresh the database portion of the Object Explorer pane to add the **Query Store** section.



Select **Regressed Queries** to open the **Regressed Queries** pane in Management Studio. The Regressed Queries pane shows you the queries and plans in the query store. Use the drop down boxes at the top to select queries based on various criteria. Select a plan to see the graphical query plan. Buttons are available to view the source query, force, and unforce a query plan, and refresh the display.

TOP 25 REGRESSED QUERIES DURING THE LAST HOUR FOR DATABASE ADVENTUREWORKS2012

query id	query sql	Duration	▼ regr perc recent	avg duration recent	avg duration hist	exec count recent	exec count hist	num plans
31	SELECT ...	CPU Time	0	26002	26002	1	1	1
29	(@_msps ...)	Logical Reads	0	33652.15	33652.15	34	34	1
28	(@_msps ...)	Logical Writes	0	14667.33	14667.33	3	3	1
27	(@_msps ...)	Physical Reads	0	6000	6000	1	1	1
26	(@_msps ...)	Memory Consumption	0	17992	17992	1	1	1
25	SELECT	0	20003	20003	1	1	1

Plan Summary For Query 31

plan id	exec count	min duration	max duration	avg duration	std dev duration	last duration	first exec time	last exec time	plan forced
1	1	26002	26002	26002	0	26002	2014-10-09 21:27:15.6800000 +00:00	2014-10-09 21:27:15.6800000 +00:00	No

Query 1: Query cost (relative to the batch): 100%
 SELECT TOP (10) WITH TIES pp.FirstName, pp.LastName, e.JobTitle, e.Gender, e.Rate FROM Person.Person AS pp INNER JOIN HumanResou...

To force a plan, select a query and plan, and then click **Force Plan**. You can only force plans that were saved by the query plan feature and are still retained in the query plan cache.

Finding wait queries

Starting from SQL Server 2017 CTP 2.0 and on Azure SQL Database wait statistics per query over time are available for Query Store customers. In Query Store wait types are combined into **wait categories**. Full mapping is available here [sys.query_store_wait_stats \(Transact-SQL\)](#)

Wait categories are combining different wait types into buckets similar by nature. Different wait categories require a different follow up analysis to resolve the issue, but wait types from the same category lead to very similar troubleshooting experiences, and providing the affected query on top of waits would be the missing piece to complete the majority of such investigations successfully.

Here are some examples how you can get more insights into your workload before and after introducing wait categories in Query Store:

Previous experience	New experience	Action
High RESOURCE_SEMAPHORE waits per database	High Memory waits in Query Store for specific queries	Find the top memory consuming queries in Query Store. These queries are probably delaying further progress of the affected queries. Consider using MAX_GRANT_PERCENT query hint for these queries, or for the affected queries.
High LCK_M_X waits per database	High Lock waits in Query Store for specific queries	Check the query texts for the affected queries and identify the target entities. Look in Query Store for other queries modifying the same entity, which are executed frequently and/or have high duration. After identifying these queries, consider changing the application logic to improve concurrency, or use a less restrictive isolation level.
High PAGEIOLATCH_SH waits per database	High Buffer IO waits in Query Store for specific queries	Find the queries with a high number of physical reads in Query Store. If they match the queries with high IO waits, consider introducing an index on the underlying entity, in order to do seeks instead of scans, and thus minimize the IO overhead of the queries.
High SOS_SCHEDULER_YIELD waits per database	High CPU waits in Query Store for specific queries	Find the top CPU consuming queries in Query Store. Among them, identify the queries for which high CPU trend correlates with high CPU waits for the affected queries. Focus on optimizing those queries – there could be a plan regression, or perhaps a missing index.

Configuration Options

The following options are available to configure query store parameters.

`OPERATION_MODE`

Can be **READ_WRITE** (default) or **READ_ONLY**.

`CLEANUP_POLICY (STALE_QUERY_THRESHOLD_DAYS)`

Configure the `STALE_QUERY_THRESHOLD_DAYS` argument to specify the number of days to retain data in the query store. The default value is 30. For SQL Database Basic edition, default is **7** days.

`DATA_FLUSH_INTERVAL_SECONDS`

Determines the frequency at which data written to the query store is persisted to disk. To optimize for performance, data collected by the query store is asynchronously written to the disk. The frequency at which this asynchronous transfer occurs is configured via `DATA_FLUSH_INTERVAL_SECONDS`. The default value is **900** (15 min).

`MAX_STORAGE_SIZE_MB`

Configures the maximum size of the query store. If the data in the query store hits the `MAX_STORAGE_SIZE_MB` limit, the query store automatically changes the state from read-write to read-only and stops collecting new data.

The default value is 100Mb. For SQL Database Premium edition, default is **1 GB** and for SQL Database Basic edition, default is **10 MB**.

INTERVAL_LENGTH_MINUTES

Determines the time interval at which runtime execution statistics data is aggregated into the query store. To optimize for space usage, the runtime execution statistics in the Runtime Stats Store are aggregated over a fixed time window. This fixed time window is configured via INTERVAL_LENGTH_MINUTES. The default value is **60**.

SIZE_BASED_CLEANUP_MODE

Controls whether the cleanup process will be automatically activated when total amount of data gets close to maximum size. Can be **AUTO** (default) or OFF.

QUERY_CAPTURE_MODE

Designates if the Query Store captures all queries, or relevant queries based on execution count and resource consumption, or stops adding new queries and just tracks current queries. Can be ALL (capture all queries), AUTO (ignore infrequent and queries with insignificant compile and execution duration) or NONE (stop capturing new queries). The default value on SQL Server (from SQL Server 2016 to SQL Server 2017) is **ALL**, while on Azure SQL Database is AUTO.

MAX_PLANS_PER_QUERY

An integer representing the maximum number of plans maintained for each query. The default value is **200**.

WAIT_STATS_CAPTURE_MODE

Controls if Query Store captures wait statistics information. Can be OFF or **ON** (default).

Query the **sys.database_query_store_options** view to determine the current options of the query store. For more information about the values, see [sys.database_query_store_options](#).

For more information about setting options by using Transact-SQL statements, see [Option Management](#).

Related Views, Functions, and Procedures

View and manage Query Storethrough Management Studio or by using the following views and procedures.

Query Store Functions

Functions help operations with the Query Store.

sys.fn_stmt_sql_handle_from_sql_stmt (Transact-SQL)	

Query Store Catalog Views

Catalog views present information about the Query Store.

sys.database_query_store_options (Transact-SQL)	sys.query_context_settings (Transact-SQL)
sys.query_store_plan (Transact-SQL)	sys.query_store_query (Transact-SQL)
sys.query_store_query_text (Transact-SQL)	sys.query_store_runtime_stats (Transact-SQL)
sys.query_store_wait_stats (Transact-SQL)	sys.query_store_runtime_stats_interval (Transact-SQL)

Query Store Stored Procedures

Stored procedures configure the Query Store.

sp_query_store_flush_db (Transact-SQL)	sp_query_store_reset_exec_stats (Transact-SQL)
sp_query_store_force_plan (Transact-SQL)	sp_query_store_unforce_plan (Transact-SQL)
sp_query_store_remove_plan (Transact-SQL)	sp_query_store_remove_query (Transact-SQL)

Key Usage Scenarios

Option Management

This section provides some guidelines on managing Query Store feature itself.

Is Query Store currently active?

Query Store stores its data inside the user database and that is why it has size limit (configured with `MAX_STORAGE_SIZE_MB`). If data in Query Store hits that limit Query Store will automatically change state from read-write to read-only and stop collecting new data.

Query [sys.database_query_store_options](#) to determine if Query Store is currently active, and whether it is currently collects runtime stats or not.

```
SELECT actual_state, actual_state_desc, readonly_reason,
       current_storage_size_mb, max_storage_size_mb
FROM sys.database_query_store_options;
```

Query Store status is determined by `actual_state` column. If it's different than the desired status, the `readonly_reason` column can give you more information.

When Query Store size exceeds the quota, the feature will switch to `readon_only` mode.

Get Query Store options

To find out detailed information about Query Store status, execute following in a user database.

```
SELECT * FROM sys.database_query_store_options;
```

Setting Query Store interval

You can override interval for aggregating query runtime statistics (default is 60 minutes).

```
ALTER DATABASE <database_name>
SET QUERY_STORE (INTERVAL_LENGTH_MINUTES = 15);
```

NOTE

Arbitrary values are not allowed for `INTERVAL_LENGTH_MINUTES`. Use one of the following: 1, 5, 10, 15, 30, 60, or 1440 minutes.

New value for interval is exposed through [sys.database_query_store_options](#) view.

Query Store space usage

To check current the Query Store size and limit execute the following statement in the user database.

```
SELECT current_storage_size_mb, max_storage_size_mb
FROM sys.database_query_store_options;
```

If the Query Store storage is full use the following statement to extend the storage.

```
ALTER DATABASE <database_name>
SET QUERY_STORE (MAX_STORAGE_SIZE_MB = <new_size>);
```

Set all Query Store options

You can set multiple Query Store options at once with a single ALTER DATABASE statement.

```
ALTER DATABASE <database name>
SET QUERY_STORE (
    OPERATION_MODE = READ_WRITE,
    CLEANUP_POLICY = (STALE_QUERY_THRESHOLD_DAYS = 30),
    DATA_FLUSH_INTERVAL_SECONDS = 3000,
    MAX_STORAGE_SIZE_MB = 500,
    INTERVAL_LENGTH_MINUTES = 15,
    SIZE_BASED_CLEANUP_MODE = AUTO,
    QUERY_CAPTURE_MODE = AUTO,
    MAX_PLANS_PER_QUERY = 1000,
    WAIT_STATS_CAPTURE_MODE = ON
);
```

Cleaning up the space

Query Store internal tables are created in the PRIMARY filegroup during database creation and that configuration cannot be changed later. If you are running out of space you might want to clear older Query Store data by using the following statement.

```
ALTER DATABASE <db_name> SET QUERY_STORE CLEAR;
```

Alternatively, you might want to clear up only ad-hoc query data, since it is less relevant for query optimizations and plan analysis but takes up just as much space.

Delete ad-hoc queries This deletes the queries that were only executed only once and that are more than 24 hours old.

```

DECLARE @id int
DECLARE adhoc_queries_cursor CURSOR
FOR
SELECT q.query_id
FROM sys.query_store_query_text AS qt
JOIN sys.query_store_query AS q
    ON q.query_text_id = qt.query_text_id
JOIN sys.query_store_plan AS p
    ON p.query_id = q.query_id
JOIN sys.query_store_runtime_stats AS rs
    ON rs.plan_id = p.plan_id
GROUP BY q.query_id
HAVING SUM(rs.count_executions) < 2
AND MAX(rs.last_execution_time) < DATEADD (hour, -24, GETUTCDATE())
ORDER BY q.query_id ;

OPEN adhoc_queries_cursor ;
FETCH NEXT FROM adhoc_queries_cursor INTO @id;
WHILE @@fetch_status = 0
BEGIN
    PRINT @id
    EXEC sp_query_store_remove_query @id
    FETCH NEXT FROM adhoc_queries_cursor INTO @id
END
CLOSE adhoc_queries_cursor ;
DEALLOCATE adhoc_queries_cursor;

```

You can define your own procedure with different logic for clearing up data you no longer want.

The example above uses the **sp_query_store_remove_query** extended stored procedure for removing unnecessary data. You can also use:

- **sp_query_store_reset_exec_stats** – to clear runtime statistics for a given plan.
- **sp_query_store_remove_plan** – to remove a single plan.

Performance Auditing and Troubleshooting

Query Store keeps a history of compilation and runtime metrics throughout query executions, allowing you to ask questions about your workload.

Last *n* queries executed on the database?

```

SELECT TOP 10 qt.query_sql_text, q.query_id,
    qt.query_text_id, p.plan_id, rs.last_execution_time
FROM sys.query_store_query_text AS qt
JOIN sys.query_store_query AS q
    ON qt.query_text_id = q.query_text_id
JOIN sys.query_store_plan AS p
    ON q.query_id = p.query_id
JOIN sys.query_store_runtime_stats AS rs
    ON p.plan_id = rs.plan_id
ORDER BY rs.last_execution_time DESC;

```

Number of executions for each query?

```

SELECT q.query_id, qt.query_text_id, qt.query_sql_text,
       SUM(rs.count_executions) AS total_execution_count
FROM sys.query_store_query_text AS qt
JOIN sys.query_store_query AS q
      ON qt.query_text_id = q.query_text_id
JOIN sys.query_store_plan AS p
      ON q.query_id = p.query_id
JOIN sys.query_store_runtime_stats AS rs
      ON p.plan_id = rs.plan_id
GROUP BY q.query_id, qt.query_text_id, qt.query_sql_text
ORDER BY total_execution_count DESC;

```

The number of queries with the longest average execution time within last hour?

```

SELECT TOP 10 rs.avg_duration, qt.query_sql_text, q.query_id,
              qt.query_text_id, p.plan_id, GETUTCDATE() AS CurrentUTCTime,
              rs.last_execution_time
FROM sys.query_store_query_text AS qt
JOIN sys.query_store_query AS q
      ON qt.query_text_id = q.query_text_id
JOIN sys.query_store_plan AS p
      ON q.query_id = p.query_id
JOIN sys.query_store_runtime_stats AS rs
      ON p.plan_id = rs.plan_id
WHERE rs.last_execution_time > DATEADD(hour, -1, GETUTCDATE())
ORDER BY rs.avg_duration DESC;

```

The number of queries that had the biggest average physical IO reads in last 24 hours, with corresponding average row count and execution count?

```

SELECT TOP 10 rs.avg_physical_io_reads, qt.query_sql_text,
              q.query_id, qt.query_text_id, p.plan_id, rs.runtime_stats_id,
              rsi.start_time, rsi.end_time, rs.avg_rowcount, rs.count_executions
FROM sys.query_store_query_text AS qt
JOIN sys.query_store_query AS q
      ON qt.query_text_id = q.query_text_id
JOIN sys.query_store_plan AS p
      ON q.query_id = p.query_id
JOIN sys.query_store_runtime_stats AS rs
      ON p.plan_id = rs.plan_id
JOIN sys.query_store_runtime_stats_interval AS rsi
      ON rsi.runtime_stats_interval_id = rs.runtime_stats_interval_id
WHERE rsi.start_time >= DATEADD(hour, -24, GETUTCDATE())
ORDER BY rs.avg_physical_io_reads DESC;

```

Queries with multiple plans? These queries are especially interesting because they are candidates for regressions due to plan choice change. The following query identifies these queries along with all plans:


```

WITH Query_MultPlans
AS
(
SELECT COUNT(*) AS cnt, q.query_id
FROM sys.query_store_query_text AS qt
JOIN sys.query_store_query AS q
    ON qt.query_text_id = q.query_text_id
JOIN sys.query_store_plan AS p
    ON p.query_id = q.query_id
GROUP BY q.query_id
HAVING COUNT(distinct plan_id) > 1
)

SELECT q.query_id, object_name(object_id) AS ContainingObject,
    query_sql_text, plan_id, p.query_plan AS plan_xml,
    p.last_compile_start_time, p.last_execution_time
FROM Query_MultPlans AS qm
JOIN sys.query_store_query AS q
    ON qm.query_id = q.query_id
JOIN sys.query_store_plan AS p
    ON q.query_id = p.query_id
JOIN sys.query_store_query_text qt
    ON qt.query_text_id = q.query_text_id
ORDER BY query_id, plan_id;

```

Queries that recently regressed in performance (comparing different point in time)? The following query example returns all queries for which execution time doubled in last 48 hours due to a plan choice change. Query compares all runtime stat intervals side by side.

```

SELECT
    qt.query_sql_text,
    q.query_id,
    qt.query_text_id,
    rs1.runtime_stats_id AS runtime_stats_id_1,
    rsi1.start_time AS interval_1,
    p1.plan_id AS plan_1,
    rs1.avg_duration AS avg_duration_1,
    rs2.avg_duration AS avg_duration_2,
    p2.plan_id AS plan_2,
    rsi2.start_time AS interval_2,
    rs2.runtime_stats_id AS runtime_stats_id_2
FROM sys.query_store_query_text AS qt
JOIN sys.query_store_query AS q
    ON qt.query_text_id = q.query_text_id
JOIN sys.query_store_plan AS p1
    ON q.query_id = p1.query_id
JOIN sys.query_store_runtime_stats AS rs1
    ON p1.plan_id = rs1.plan_id
JOIN sys.query_store_runtime_stats_interval AS rsi1
    ON rsi1.runtime_stats_interval_id = rs1.runtime_stats_interval_id
JOIN sys.query_store_plan AS p2
    ON q.query_id = p2.query_id
JOIN sys.query_store_runtime_stats AS rs2
    ON p2.plan_id = rs2.plan_id
JOIN sys.query_store_runtime_stats_interval AS rsi2
    ON rsi2.runtime_stats_interval_id = rs2.runtime_stats_interval_id
WHERE rsi1.start_time > DATEADD(hour, -48, GETUTCDATE())
    AND rsi2.start_time > rsi1.start_time
    AND p1.plan_id <> p2.plan_id
    AND rs2.avg_duration > 2*rs1.avg_duration
ORDER BY q.query_id, rsi1.start_time, rsi2.start_time;

```

If you want to see performance all regressions (not only those related to plan choice change) than just remove condition `AND p1.plan_id <> p2.plan_id` from the previous query.

Queries that are waiting the most? This query will return top 10 queries that wait the most.

```
SELECT TOP 10
    qt.query_text_id,
    q.query_id,
    p.plan_id,
    sum(total_query_wait_time_ms) AS sum_total_wait_ms
FROM sys.query_store_wait_stats ws
JOIN sys.query_store_plan p ON ws.plan_id = p.plan_id
JOIN sys.query_store_query q ON p.query_id = q.query_id
JOIN sys.query_store_query_text qt ON q.query_text_id = qt.query_text_id
GROUP BY qt.query_text_id, q.query_id, p.plan_id
ORDER BY sum_total_wait_ms DESC
```

Queries that recently regressed in performance (comparing recent vs. history execution)? The next query compares query execution based periods of execution. In this particular example the query compares execution in recent period (1 hour) vs. history period (last day) and identifies those that introduced `additional_duration_workload`. This metrics is calculated as a difference between recent average execution and history average execution multiplied by the number of recent executions. It actually represents how much of additional duration recent executions introduced compared to history:

```
--- "Recent" workload - last 1 hour
DECLARE @recent_start_time datetimeoffset;
DECLARE @recent_end_time datetimeoffset;
SET @recent_start_time = DATEADD(hour, -1, SYSUTCDATETIME());
SET @recent_end_time = SYSUTCDATETIME();

--- "History" workload
DECLARE @history_start_time datetimeoffset;
DECLARE @history_end_time datetimeoffset;
SET @history_start_time = DATEADD(hour, -24, SYSUTCDATETIME());
SET @history_end_time = SYSUTCDATETIME();

WITH
hist AS
(
    SELECT
        p.query_id query_id,
        CONVERT(float, SUM(rs.avg_duration*rs.count_executions)) total_duration,
        SUM(rs.count_executions) count_executions,
        COUNT(distinct p.plan_id) num_plans
    FROM sys.query_store_runtime_stats AS rs
    JOIN sys.query_store_plan p ON p.plan_id = rs.plan_id
    WHERE (rs.first_execution_time >= @history_start_time
        AND rs.last_execution_time < @history_end_time)
        OR (rs.first_execution_time <= @history_start_time
        AND rs.last_execution_time > @history_start_time)
        OR (rs.first_execution_time <= @history_end_time
        AND rs.last_execution_time > @history_end_time)
    GROUP BY p.query_id
),
recent AS
(
    SELECT
        p.query_id query_id,
        CONVERT(float, SUM(rs.avg_duration*rs.count_executions)) total_duration,
        SUM(rs.count_executions) count_executions,
        COUNT(distinct p.plan_id) num_plans
    FROM sys.query_store_runtime_stats AS rs
    JOIN sys.query_store_plan p ON p.plan_id = rs.plan_id
    WHERE (rs.first_execution_time >= @recent_start_time
        AND rs.last_execution_time < @recent_end_time)
        OR (rs.first_execution_time <= @recent_start_time
        AND rs.last_execution_time > @recent_start_time)
```

```

        OR (rs.first_execution_time \<= @recent_end_time
            AND rs.last_execution_time > @recent_end_time)
    GROUP BY p.query_id
)
SELECT
    results.query_id query_id,
    results.query_text query_text,
    results.additional_duration_workload additional_duration_workload,
    results.total_duration_recent total_duration_recent,
    results.total_duration_hist total_duration_hist,
    ISNULL(results.count_executions_recent, 0) count_executions_recent,
    ISNULL(results.count_executions_hist, 0) count_executions_hist
FROM
(
    SELECT
        hist.query_id query_id,
        qt.query_sql_text query_text,
        ROUND(CONVERT(float, recent.total_duration/
            recent.count_executions-hist.total_duration/hist.count_executions)
            *(recent.count_executions), 2) AS additional_duration_workload,
        ROUND(recent.total_duration, 2) total_duration_recent,
        ROUND(hist.total_duration, 2) total_duration_hist,
        recent.count_executions count_executions_recent,
        hist.count_executions count_executions_hist
    FROM hist
    JOIN recent
        ON hist.query_id = recent.query_id
    JOIN sys.query_store_query AS q
        ON q.query_id = hist.query_id
    JOIN sys.query_store_query_text AS qt
        ON q.query_text_id = qt.query_text_id
) AS results
WHERE additional_duration_workload > 0
ORDER BY additional_duration_workload DESC
OPTION (MERGE JOIN);

```

Maintaining Query Performance Stability

For queries executed multiple times you may notice that SQL Server uses different plans, resulting in different resource utilization and duration. With Query Store you can detect when query performance regressed and determine the optimal plan within a period of interest. You can then force that optimal plan for future query execution.

You can also identify inconsistent query performance for a query with parameters (either auto- parameterized or manually parameterized). Among different plans you can identify the plan which is fast and optimal enough for all or most of the parameter values and force that plan, keeping predictable performance for the wider set of user scenarios.

Force or a plan for a query (apply forcing policy). When a plan is forced for a certain query, every time a query comes to execution it will be executed with the plan that is forced.

```
EXEC sp_query_store_force_plan @query_id = 48, @plan_id = 49;
```

When using **sp_query_store_force_plan** you can only force plans that were recorded by Query Store as a plan for that query. In other words, the only plans available for a query are those that were already used to execute that query while Query Store was active.

Remove plan forcing for a query. To rely again on the SQL Server query optimizer to calculate the optimal query plan, use **sp_query_store_unforce_plan** to unforce the plan that was selected for the query.

```
EXEC sp_query_store_unforce_plan @query_id = 48, @plan_id = 49;
```

See Also

[Best Practice with the Query Store](#)

[Using the Query Store with In-Memory OLTP](#)

[Query Store Usage Scenarios](#)

[How Query Store Collects Data](#)

[Query Store Stored Procedures \(Transact-SQL\)](#)

[Query Store Catalog Views \(Transact-SQL\)](#)

[Monitor and Tune for Performance](#)

[Performance Monitoring and Tuning Tools](#)

[Open Activity Monitor \(SQL Server Management Studio\)](#)

[Live Query Statistics](#)





[Activity Monitor](#)

[sys.database_query_store_options \(Transact-SQL\)](#)

[Operating the Query Store in Azure SQL Database](#)

Using the Query Store with In-Memory OLTP

1/17/2018 • 4 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

SQL Server Query Store allows you to monitor the performance of natively compiled code for workloads running in-memory OLTP.

Compile and runtime statistics are collected and exposed the same way as for disk-based workloads.

When you migrate to in-memory OLTP you can continue using Query Store views in SQL Server Management Studio as well as custom scripts you have developed for disk-based workloads before migration. This saves your investment in learning Query Store technology and makes it generally usable for troubleshooting all type of workloads.

For general information on using the Query Store, see [Monitoring Performance By Using the Query Store](#).

Using the Query Store with in-memory OLTP does not require any additional feature configuration. When you enable it on your database it will work for all types of workloads.

However, there are some specific aspects that users should be aware of when using Query Store with in-memory OLTP:

- When Query Store is enabled, queries, plans and compile-time statistics are collected by default. However, runtime statistics collection is not activated unless you explicitly enable it with [sys.sp_xtp_control_query_exec_stats \(Transact-SQL\)](#).
- When you set `@new_collection_value` to 0 Query Store will stop collecting runtime statistics for affected procedure or for the entire SQL Server instance.
- The value configured with [sys.sp_xtp_control_query_exec_stats \(Transact-SQL\)](#) is not persisted. Make sure you check and configure again statistics collection after restarting SQL Server.
- As in case with regular query statistics collection, performance may decrease when you use Query Store to track workload execution. You may want to consider enabling statistics collection only for an important subset of natively compiled stored procedures.
- Queries and plans are captured and stored on the first native compilation and updated upon every recompilation.
- If you enabled Query Store or cleared its content after all native stored procedure were compiled you must recompile them manually in order to make them captured by the Query Store. The same applies if you removed queries manually by using [sp_query_store_remove_query \(Transact-SQL\)](#) or [sp_query_store_remove_plan \(Transact-SQL\)](#). Use [sp_recompile \(Transact-SQL\)](#) to force procedure recompilation.
- Query Store leverages plan generation mechanisms from in-memory OLTP to capture query execution plan during the compilation. Stored plan is semantically equivalent to one that you would get by using `SET SHOWPLAN_XML ON` with one difference; plans in Query Store are split and stored per individual statement.
- When you run Query Store in a database with a mixed workload then you can use **is_natively_compiled** field from [sys.query_store_plan \(Transact-SQL\)](#) to quickly find query plans that are generated by the native code compilation.
- Query Store capture mode (`QUERY_CAPTURE_MODE` parameter in **ALTER TABLE** statement) does not affect queries from natively compiled modules as they are always captured regardless of the configured value.

This includes setting `QUERY_CAPTURE_MODE = NONE`.

- The duration of query compilation captured by the Query Store includes only time spent in query optimization, before the native code was generated. More precisely, it doesn't include time for C code compilation and generation of internal structures necessary for C code generation.
- Memory grants metrics within [sys.query_store_runtime_stats \(Transact-SQL\)](#) are not populated for natively compiled queries – their values are always 0. The memory grants columns are: `avg_query_max_used_memory`, `last_query_max_used_memory`, `min_query_max_used_memory`, `max_query_max_used_memory`, and `stdev_query_max_used_memory`.

Enabling and using Query Store with In-Memory OLTP

The following simple example demonstrates using Query Store with in-memory OLTP in an end-to-end user scenario. In this example we assume that a database (`MemoryOLTP`) is enabled for in-memory OLTP.

For more details on prerequisites for memory-optimized tables, see [Creating a Memory-Optimized Table and a Natively Compiled Stored Procedure](#).

```
USE MemoryOLTP;
GO

-- Create a simple memory-optimized table
CREATE TABLE dbo.Ord
    (OrdNo INTEGER not null PRIMARY KEY NONCLUSTERED,
    OrdDate DATETIME not null,
    CustCode NVARCHAR(5) not null)
WITH (MEMORY_OPTIMIZED=ON);
GO

-- Enable Query Store before native module compilation
ALTER DATABASE MemoryOLTP SET QUERY_STORE = ON;
GO

-- Create natively compiled stored procedure
CREATE PROCEDURE dbo.OrderInsert(@OrdNo integer, @CustCode nvarchar(5))
WITH NATIVE_COMPILATION, SCHEMABINDING
AS
    BEGIN ATOMIC WITH
        (TRANSACTION ISOLATION LEVEL = SNAPSHOT,
        LANGUAGE = N'English')

    DECLARE @OrdDate DATETIME = GETDATE();
    INSERT INTO dbo.Ord (OrdNo, CustCode, OrdDate)
    VALUES (@OrdNo, @CustCode, @OrdDate);
END;
GO

-- Enable runtime stats collection for queries from dbo.OrderInsert stored procedure
DECLARE @db_id INT = DB_ID()
DECLARE @proc_id INT = OBJECT_ID('dbo.OrderInsert');
DECLARE @collection_enabled BIT;

EXEC [sys].[sp_xtp_control_query_exec_stats] @new_collection_value = 1,
    @database_id = @db_id, @xtp_object_id = @proc_id;

-- Check the state of the collection flag
EXEC sp_xtp_control_query_exec_stats @database_id = @db_id,
    @xtp_object_id = @proc_id,
    @old_collection_value= @collection_enabled output;
SELECT @collection_enabled AS 'collection status';

-- Execute natively compiled workload
EXEC dbo.OrderInsert 1, 'A';
EXEC dbo.OrderInsert 2, 'B';
```

```

EXEC dbo.OrderInsert 3, 'C';
EXEC dbo.OrderInsert 4, 'D';
EXEC dbo.OrderInsert 5, 'E';

-- Check Query Store Data
-- Compile time data
SELECT q.query_id, plan_id, object_id, query_hash, p.query_plan,
       p.initial_compile_start_time, p.last_compile_start_time,
       p.last_execution_time, p.avg_compile_duration,
       p.last_force_failure_reason, p.force_failure_count
FROM sys.query_store_query AS q
JOIN sys.query_store_plan AS p
     ON q.query_id = p.plan_id
WHERE q.object_id = OBJECT_ID('dbo.OrderInsert');

-- Get runtime stats
-- Check count_executions field to verify that runtime statistics
-- have been collected by the Query Store
SELECT q.query_id, p.plan_id, object_id, rsi.start_time, rsi.end_time,
       p.last_force_failure_reason, p.force_failure_count, rs.*
FROM sys.query_store_query AS q
JOIN sys.query_store_plan AS p
     ON q.query_id = p.plan_id
JOIN sys.query_store_runtime_stats AS rs
     ON rs.plan_id = p.plan_id
JOIN sys.query_store_runtime_stats_interval AS rsi
     ON rs.runtime_stats_interval_id = rsi.runtime_stats_interval_id
WHERE q.object_id = OBJECT_ID('dbo.OrderInsert');

```

See Also

[Monitoring Performance By Using the Query Store](#)

[Creating a Memory-Optimized Table and a Natively Compiled Stored Procedure](#)





[Best Practice with the Query Store](#)

[Query Store Stored Procedures \(Transact-SQL\)](#)

[Query Store Catalog Views \(Transact-SQL\)](#)

Best Practice with the Query Store

1/17/2018 • 14 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

This topic outlines the best practices for using the Query Store with your workload.

Use the latest SQL Server Management Studio

SQL Server Management Studio has set of user interfaces designed for configuring Query Store as well as for consuming collected data about your workload.

Download the latest version of Management Studio [here](#).

For a quick description on how to use Query Store in troubleshooting scenarios refer to [Query Store @Azure Blogs](#).

Use Query Performance Insight in Azure SQL Database

If you run Query Store in SQL Database you can use **Query Performance Insight** to analyze DTU consumption over time.

While you can use Management Studio to get detailed resource consumption for all your queries (CPU, memory, IO, etc.), Query Performance Insight gives you a quick and efficient way to determine their impact on overall DTU consumption for your database.

For more information, see [Azure SQL Database Query Performance Insight](#).

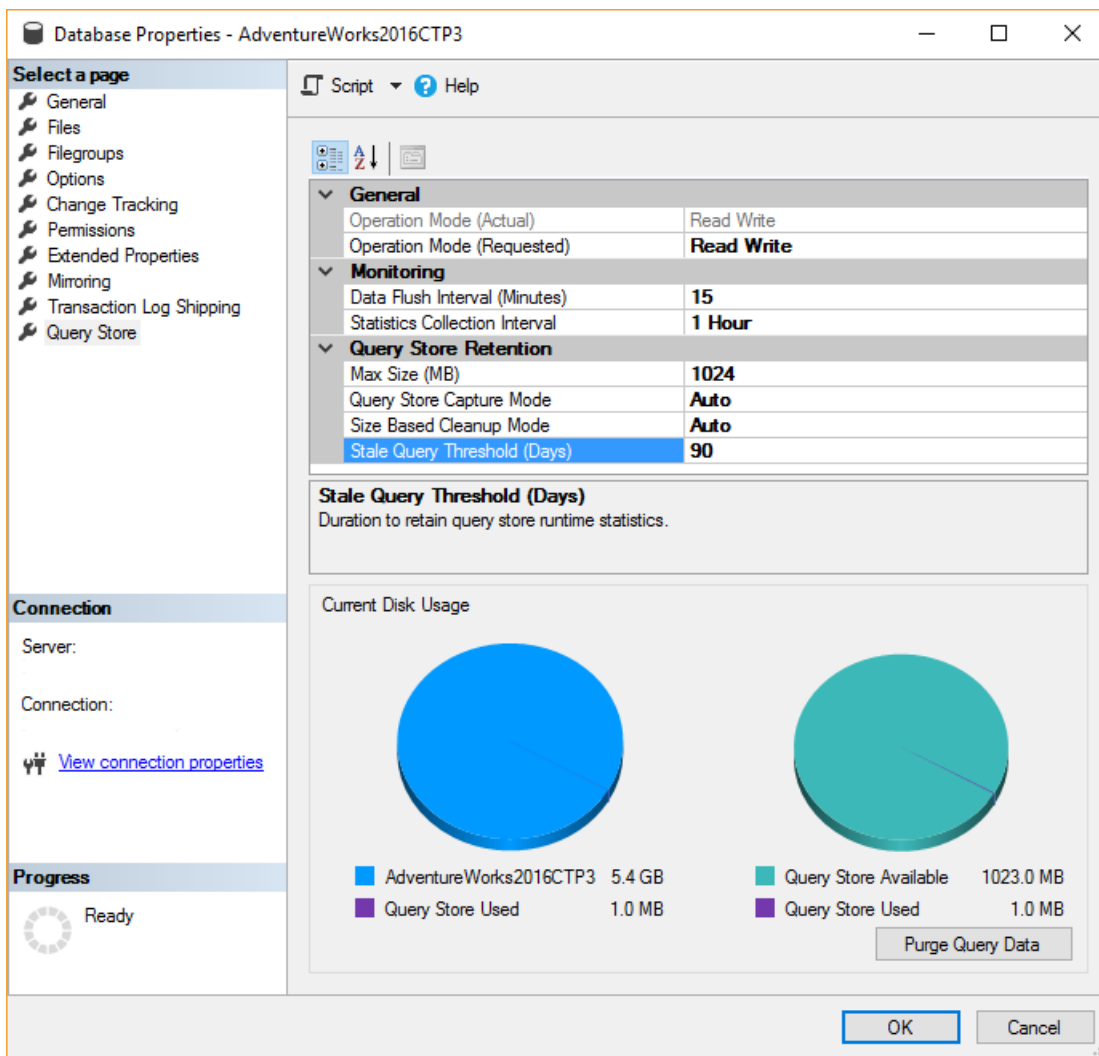
Using Query Store with Elastic Pool Databases

You can use Query Store in all databases without concerns, in even densely packed pools. All issues related to excessive resource usage, that might have occurred when Query Store was enabled for the large number of databases in the Elastic Pools, have been resolved.

Keep Query Store adjusted to your workload

Configure Query Store based on your workload and performance troubleshooting requirements.

The default parameters are good for a quick start but you should monitor how Query Store behaves over time and adjust its configuration accordingly:



Here are guidelines to follow for setting parameter values:

Max Size (MB): Specifies the limit for the data space that Query Store will take inside your database. This is the most important setting that directly affects operation mode of the Query Store.

While Query Store collects queries, execution plans and statistics, its size in the database grows until this limit is reached. When that happens, Query Store automatically changes the operation mode to read-only and stops collecting new data, which means that your performance analysis is no longer accurate.

The default value (100 MB) may not be sufficient if your workload generates large number of different queries and plans or if you want to keep query history for a longer period of time. Keep track of current space usage and increase the Max Size (MB) to prevent Query Store from transitioning to read-only mode. Use Management Studio or execute the following script to get the latest information about Query Store size:

```
USE [QueryStoreDB];
GO

SELECT actual_state_desc, desired_state_desc, current_storage_size_mb,
       max_storage_size_mb, readonly_reason
FROM sys.database_query_store_options;
```

The following script sets a new Max Size (MB):

```
ALTER DATABASE [QueryStoreDB]
SET QUERY_STORE (MAX_STORAGE_SIZE_MB = 1024);
```

Statistics Collection Interval: Defines level of granularity for the collected runtime statistic (the default is 1

hour). Consider using lower value if you require finer granularity or less time to detect and mitigate issues but keep in mind that it will directly affect the size of Query Store data. Use SSMS or Transact-SQL to set different value for Statistics Collection Interval:

```
ALTER DATABASE [QueryStoreDB] SET QUERY_STORE (INTERVAL_LENGTH_MINUTES = 60);
```

Stale Query Threshold (Days): Time-based cleanup policy that controls the retention period of persisted runtime statistics and inactive queries.

By default, Query Store is configured to keep the data for 30 days which may be unnecessarily long for your scenario.

Avoid keeping historical data that you do not plan to use. This will reduce changes to read-only status. The size of Query Store data as well as the time to detect and mitigate the issue will be more predictable. Use Management Studio or the following script to configure time-based cleanup policy:

```
ALTER DATABASE [QueryStoreDB]
SET QUERY_STORE (CLEANUP_POLICY = (STALE_QUERY_THRESHOLD_DAYS = 90));
```

Size Based Cleanup Mode: Specifies whether automatic data cleanup will take place when Query Store data size approaches the limit.

It is strongly recommended to activate size-based cleanup to make sure that Query Store always runs in read-write mode and collects the latest data.

```
ALTER DATABASE [QueryStoreDB]
SET QUERY_STORE (SIZE_BASED_CLEANUP_MODE = AUTO);
```

Query Store Capture Mode: Specifies the query capture policy for the Query Store.

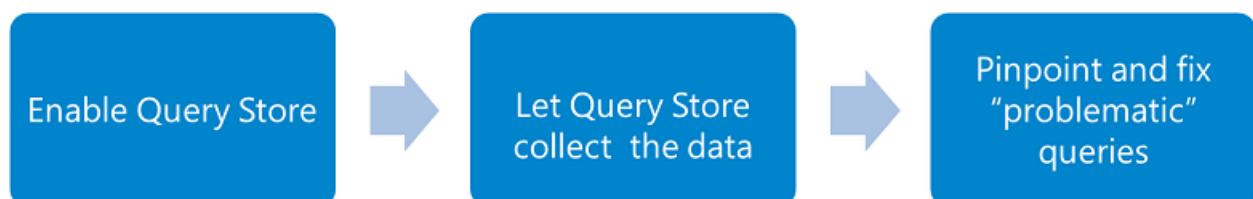
- **All** – Captures all queries. This is the default option.
- **Auto** – Infrequent queries and queries with insignificant compile and execution duration are ignored. Thresholds for execution count, compile and runtime duration are internally determined.
- **None** – Query Store stops capturing new queries.

The following script sets the Query Capture mode to Auto:

```
ALTER DATABASE [QueryStoreDB]
SET QUERY_STORE (QUERY_CAPTURE_MODE = AUTO);
```

How to start with query performance troubleshooting

Troubleshooting workflow with Query Store is simple, as shown on the following diagram:



Enable Query Store by using Management Studio as described in the previous section, or execute the following Transact-SQL statement:

```
ALTER DATABASE [DatabaseOne] SET QUERY_STORE = ON;
```

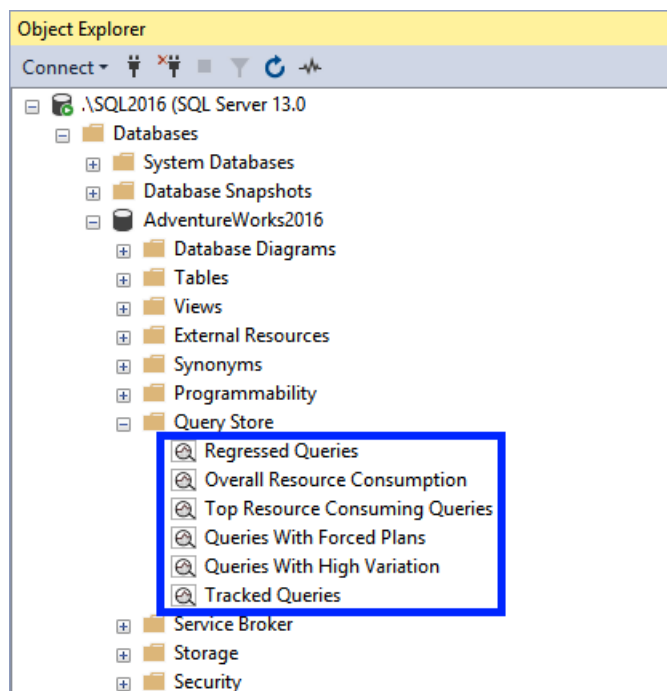
It will take some time until Query Store collects the data set that accurately represents your workload. Usually, one day is enough even for very complex workloads. However, you can start exploring the data and identify queries that need your attention immediately after you enabled the feature.

Navigate to the Query Store sub-folder under the database node in Object Explorer of Management Studio to open troubleshooting views for specific scenarios.

Management Studio Query Store views operate with the set of execution metrics, each expressed as any of the following statistic functions:

SQL SERVER VERSION	EXECUTION METRIC	STATISTIC FUNCTION
SQL Server 2016	CPU time, Duration, Execution Count, Logical Reads, Logical writes, Memory consumption, Physical Reads, CLR time, Degree of Parallelism (DOP), and Row count	Average, Maximum, Minimum, Standard Deviation, Total
SQL Server 2017	CPU time, Duration, Execution Count, Logical Reads, Logical writes, Memory consumption, Physical Reads, CLR time, Degree of Parallelism (DOP), Row count, Log memory, TempDB memory, and Wait times	Average, Maximum, Minimum, Standard Deviation, Total

The following graphic shows how to locate Query Store views:



The following table explains when to use each of the Query Store views:

SSMS VIEW	SCENARIO
Regressed Queries	Pinpoint queries for which execution metrics have recently regressed (i.e. changed to worse). Use this view to correlate observed performance problems in your application with the actual queries that needs to be fixed or improved.

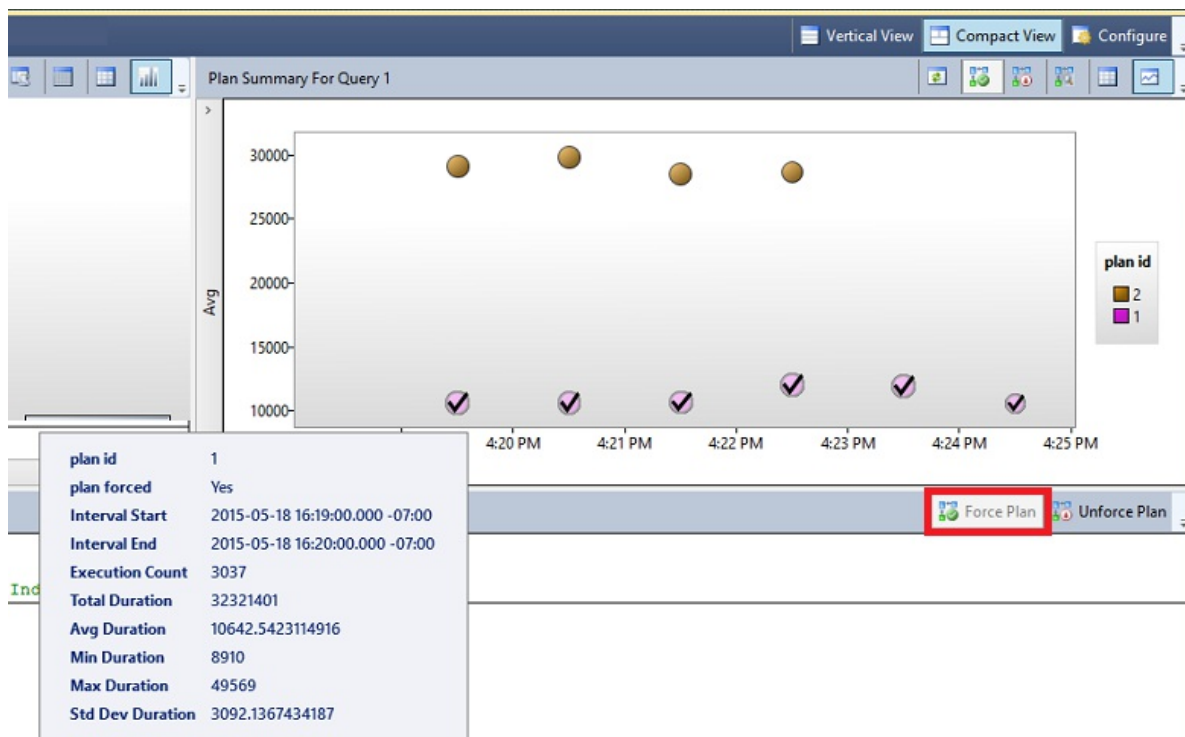
SSMS VIEW	SCENARIO
Overall Resource Consumption	Analyze the total resource consumption for the database for any of the execution metrics. Use this view to identify resource patterns (daily vs. nightly workloads) and optimize overall consumption for your database.
Top Resource Consuming Queries	Choose an execution metric of interest and identify queries that had the most extreme values for a provided time interval. Use this view to focus your attention on the most relevant queries which have the biggest impact to database resource consumption.
Queries With Forced Plans	Lists previously forced plans using Query Store. Use this view to quickly access all currently forced plans.
Queries With High Variation	Analyze queries with high execution variation as it relates to any of the available dimensions, such as Duration, CPU time, IO, and Memory usage in the desired time interval. Use this view to identify queries with widely variant performance that can be impacting user experience across your applications.
Tracked Queries	Track the execution of the most important queries in real-time. Typically, you use this view when you have queries with forced plans and you want to make sure that query performance is stable.

TIP

For a detailed description how to use Management Studio to identify the top resource consuming queries and fix those that regressed due to the change of a plan choice, see [Query Store @Azure Blogs](#).

When you identify a query with sub-optimal performance, your action depends on the nature of the problem.

- If the query was executed with multiple plans and the last plan is significantly worse than previous plan, you can use the plan forcing mechanism to force SQL Server to always use the optimal plan for future executions.



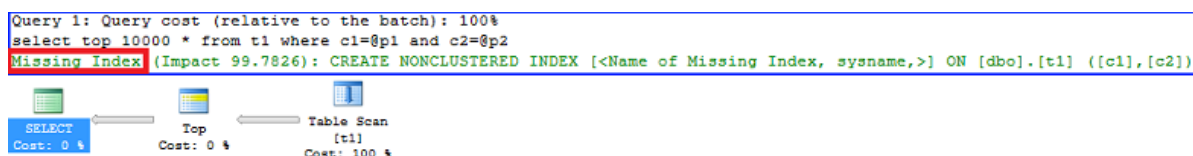
NOTE

The above graphic may feature different shapes for specific query plans, with the following meanings for each possible status:

SHAPE	MEANING
Circle	Query Completed (Regular Execution successfully finished)
Square	Cancelled (Client initiated aborted execution)
Triangle	Failed (Exception aborted execution)

Also, the size of the shape reflects query execution count within the specified time interval, increasing in size with a higher number of executions.

- You may conclude that your query is missing an index for optimal execution. This information is surfaced within the query execution plan. Create the missing index and check the query performance using the Query Store.



If you run your workload on SQL Database, sign up for SQL Database Index Advisor to automatically receive index recommendations.

- In some cases you may enforce statistic recompilation if you see that the difference between the estimated and the actual number of rows in the execution plan is significant.
- Rewrite problematic queries. For example to take advantages of query parameterization or to implement more optimal logic.

Verify Query Store is collecting query data continuously

Query Store can silently change operations mode. You should regularly monitor the state of the Query Store to ensure that the Query Store is operating, and to take action to avoid failures due to preventable causes. Execute the following query to determine the operation mode and view the most relevant parameters:

```
USE [QueryStoreDB];
GO

SELECT actual_state_desc, desired_state_desc, current_storage_size_mb,
       max_storage_size_mb, readonly_reason, interval_length_minutes,
       stale_query_threshold_days, size_based_cleanup_mode_desc,
       query_capture_mode_desc
FROM sys.database_query_store_options;
```

The difference between the `actual_state_desc` and `desired_state_desc` indicates that a change of operations mode occurred automatically. The most common change is for the Query Store to silently switch to read-only mode. In extremely rarely circumstances, Query Store can end up in the ERROR state because of internal errors.

When the actual state is read-only, use the **readonly_reason** column to determine the root cause. Typically you will find that Query Store transitioned to read-only mode because the size quota was exceeded. In that case the **readonly_reason** is set to 65536. For other reasons, see [sys.database_query_store_options \(Transact-SQL\)](#).

Consider the following steps to switch Query Store to read-write mode and activate data collection:

- Increase the maximum storage size by using the **MAX_STORAGE_SIZE_MB** option of **ALTER DATABASE**.
- Clean up Query Store data by using the following statement:

```
ALTER DATABASE [QueryStoreDB] SET QUERY_STORE CLEAR;
```

You can apply one or both of these steps by the executing the following statement that explicitly changes operation mode back to read-write:

```
ALTER DATABASE [QueryStoreDB]
SET QUERY_STORE (OPERATION_MODE = READ_WRITE);
```

Take the following steps to be proactive:

- You can prevent silent changes of operation mode by applying best practices. If you ensure that Query Store size is always below the maximally allowed value that will dramatically reduce a chance of transitioning to read-only mode. Activate size based policy as described in the [Configure Query Store](#) section, so that the Query Store automatically cleans data when the size approaches the limit.
- In order to make sure that most recent data is retained, configure time-based policy to remove stale information regularly.
- Finally, you should consider setting Query Capture Mode to Auto as it filters out queries that are usually less relevant for your workload.

Error State

To recover Query Store try explicitly setting the read-write mode and check actual state again.

```

ALTER DATABASE [QueryStoreDB]
SET QUERY_STORE (OPERATION_MODE = READ_WRITE);
GO

SELECT actual_state_desc, desired_state_desc, current_storage_size_mb,
       max_storage_size_mb, readonly_reason, interval_length_minutes,
       stale_query_threshold_days, size_based_cleanup_mode_desc,
       query_capture_mode_desc
FROM sys.database_query_store_options;

```

If the problem persists, it indicates corruption of the Query Store data is persisted on the disk.

Query Store could be recovered by executing **sp_query_store_consistency_check** stored procedure within the affected database.

If that did not help, you can try to clear Query Store before requesting read-write mode.

```

ALTER DATABASE [QueryStoreDB]
SET QUERY_STORE CLEAR;
GO

ALTER DATABASE [QueryStoreDB]
SET QUERY_STORE (OPERATION_MODE = READ_WRITE);
GO

SELECT actual_state_desc, desired_state_desc, current_storage_size_mb,
       max_storage_size_mb, readonly_reason, interval_length_minutes,
       stale_query_threshold_days, size_based_cleanup_mode_desc,
       query_capture_mode_desc
FROM sys.database_query_store_options;

```

Set the optimal query capture mode

Keep the most relevant data in Query Store. The following table describes typical scenarios for each Query Capture Mode:

QUERY CAPTURE MODE	SCENARIO
All	<p>Analyze your workload thoroughly in terms of all queries shapes and their execution frequencies and other statistics.</p> <p>Identify new queries in your workload.</p> <p>Detect if ad-hoc queries are used to identify opportunities for user or auto parameterization.</p>
Auto	<p>Focus your attention on relevant and actionable queries; those queries that execute regularly or that have significant resource consumption.</p>

QUERY CAPTURE MODE	SCENARIO
None	<p>You have already captured the query set that you want to monitor in runtime and you want to eliminate the distractions that other queries may introduce.</p> <p>None is suitable for testing and bench-marking environments.</p> <p>None is also appropriate for software vendors who ship Query Store configuration configured to monitor their application workload.</p> <p>None should be used with caution as you might miss the opportunity to track and optimize important new queries. Avoid using None unless you have a specific scenario that requires it.</p>

Keep the most relevant data in Query Store

Configure the Query Store to contain only the relevant data and it will run continuously providing great troubleshooting experience with a minimal impact on your regular workload.

The following table provides best practices:

BEST PRACTICE	SETTING
Limit retained historical data.	Configure time-based policy to activate auto-cleanup.
Filter out non-relevant queries.	Configure Query Capture Mode to Auto.
Delete less relevant queries when maximum size is reached.	Activate size-based cleanup policy.

Avoid using non-parameterized queries

Using non-parameterized queries when that is not absolutely necessary (for example in case of ad-hoc analysis) is not a best practice. Cached plans cannot be reused which forces Query Optimizer to compile queries for every unique query text. For more information on this topic, see [Guidelines for Using Forced Parameterization](#).

Also, Query Store can rapidly exceed the size quota because of potentially a large number of different query texts and consequently a large number of different execution plans with similar shape.

As a result, performance of your workload will be sub-optimal and Query Store might switch to read-only mode or might be constantly deleting the data trying to keep up with the incoming queries.

Consider following options:

- Parameterize queries where applicable, for example wrap queries inside a stored procedure or `sp_executesql`. For more information on this topic, see [Parameters and Execution Plan Reuse](#).
- Use the **Optimize for Ad Hoc Workloads** option if your workload contains many single use ad-hoc batches with different query plans.
 - Compare the number of distinct `query_hash` values with the total number of entries in `sys.query_store_query`. If the ratio is close to 1 your ad-hoc workload generates different queries.
- Apply **forced parameterization**, for the database or for a subset of queries if the number of different query plans is not large.
 - Use [plan guide](#) to force parameterization only for the selected query.

- Configure forced parameterization as using the [Parameterization database option](#) command, if there are a small number of different query plans in your workload: when the ratio between the count of distinct query_hash and the total number of entries in sys.query_store_query is much less than 1.
- Set the **Query Capture Mode** to AUTO to automatically filter out ad-hoc queries with small resource consumption.

Avoid a DROP and CREATE pattern when maintaining containing objects for the queries

Query Store associates query entry with a containing object (stored procedure, function, and trigger). When you recreate a containing object, a new query entry will be generated for the same query text. This will prevent you from tracking performance statistics for that query over time and use plan forcing mechanism. To avoid this, use the `ALTER <object>` process to change a containing object definition whenever it is possible.

Check the status of Forced Plans regularly

Plan forcing is a convenient mechanism to fix performance for the critical queries and make them more predictable. However, as with plan hints and plan guides, forcing a plan is not a guarantee that it will be used in future executions. Typically, when database schema changes in a way that objects referenced by the execution plan are altered or dropped, plan forcing will start failing. In that case SQL Server falls back to query recompilation while the actual forcing failure reason is surfaced in [sys.query_store_plan](#). The following query returns information about forced plans:

```
USE [QueryStoreDB];
GO

SELECT p.plan_id, p.query_id, q.object_id as containing_object_id,
       force_failure_count, last_force_failure_reason_desc
FROM sys.query_store_plan AS p
JOIN sys.query_store_query AS q on p.query_id = q.query_id
WHERE is_forced_plan = 1;
```

For full list of reasons, refer to [sys.query_store_plan](#). You can also use the **query_store_plan_forcing_failed** XEvent to track troubleshoot plan forcing failures.

Avoid renaming databases if you have queries with Forced Plans

Execution plans reference objects using three-part names `database.schema.object`.

If you rename a database, plan forcing will fail which will cause recompilation in all subsequent query executions.

Use traceflags on mission critical servers to improve recovery from disaster

The global traceflags 7745 and 7752 can be used to improve performance of Query Store during High Availability and Disaster Recovery scenarios. For more information, refer to [Trace Flags](#)

Traceflag 7745 will prevent the default behavior where Query Store writes data to disk before SQL Server can be shutdown.

Traceflag 7752 enables asynchronous load of Query Store, and also allows SQL Server to run queries before Query Store has been fully loaded. Default Query Store behavior prevents queries from running before the Query Store has been recovered.

See Also

[Query Store Catalog Views \(Transact-SQL\)](#)

[Query Store Stored Procedures \(Transact-SQL\)](#)





[Using the Query Store with In-Memory OLTP](#)

[Monitoring Performance By Using the Query Store](#)

[Query Processing Architecture Guide](#)

Query Store Usage Scenarios

1/17/2018 • 10 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

Query Store can be used in wide set of scenarios when tracking and ensuring predictable workload performance is critical. Here are some examples you can consider:

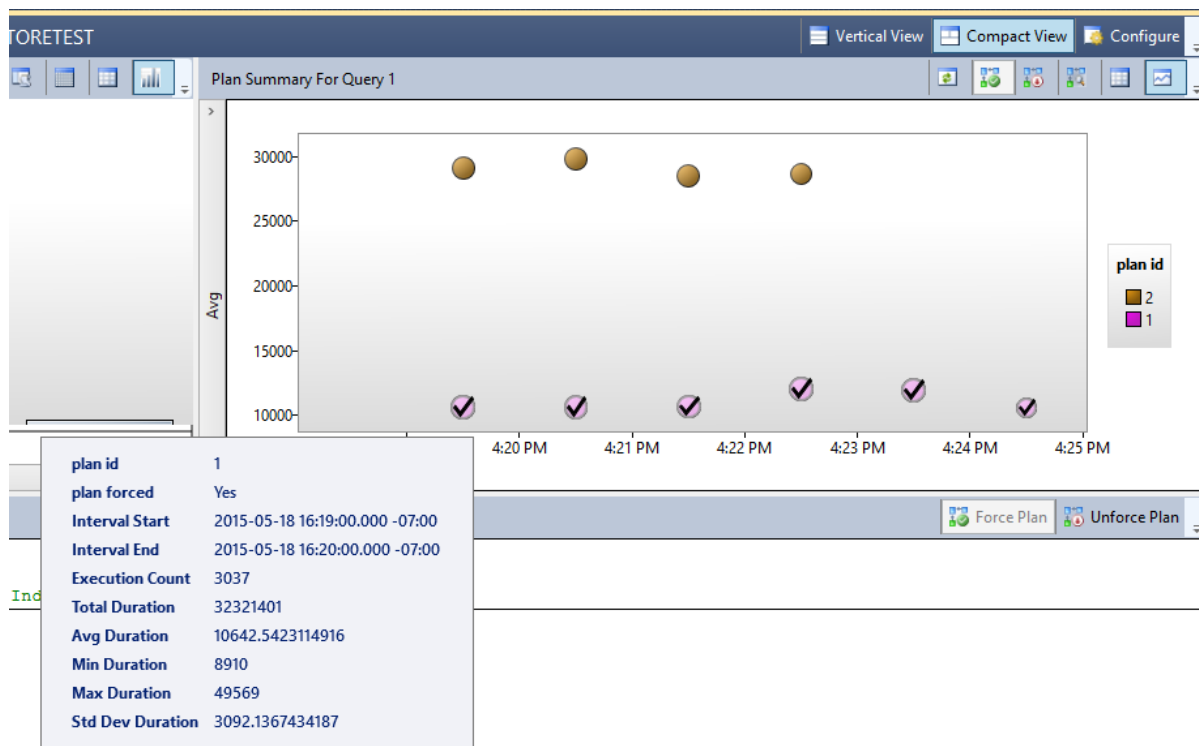
- Pinpoint and fix queries with plan choice regressions
- Identify and tune top resource consuming queries
- A/B testing
- Keep performance stability during the upgrade to newer SQL Server
- Identify and improve ad-hoc workloads

Pinpoint and fix queries with plan choice regressions

During the regular query execution Query Optimizer may decide to take a different plan because important inputs became different: data cardinality has changed, indexes have been created, altered or dropped, statistics have been updated, etc. For the most part new plan it picks is better or about the same than one was used previously. However, there are cases when new plan is significantly worse - we refer to that situation as plan choice change regression. Prior to Query Store, it was an issue very difficult to identify and fix as SQL Server didn't provide built-in data store for users to look at for execution plans that were used over time.

With the Query Store you can quickly:

- Identify all queries which execution metrics have been degraded in the period of time of interest (last hour, day, week, etc.). Use **Regressed Queries** in SQL Server Management Studio to speed up your analysis.
- Among the regressed queries it's very easy to find those that had multiple plans and which degraded because of the bad plan choice. Use **Plan Summary** pane in **Regressed Queries** to visualize all plans for a regressed query and their query performance over time.
- Force the previous plan from the history if it proved to be better. Use **Force Plan** button in **Regressed Queries** to force selected plan for the query.

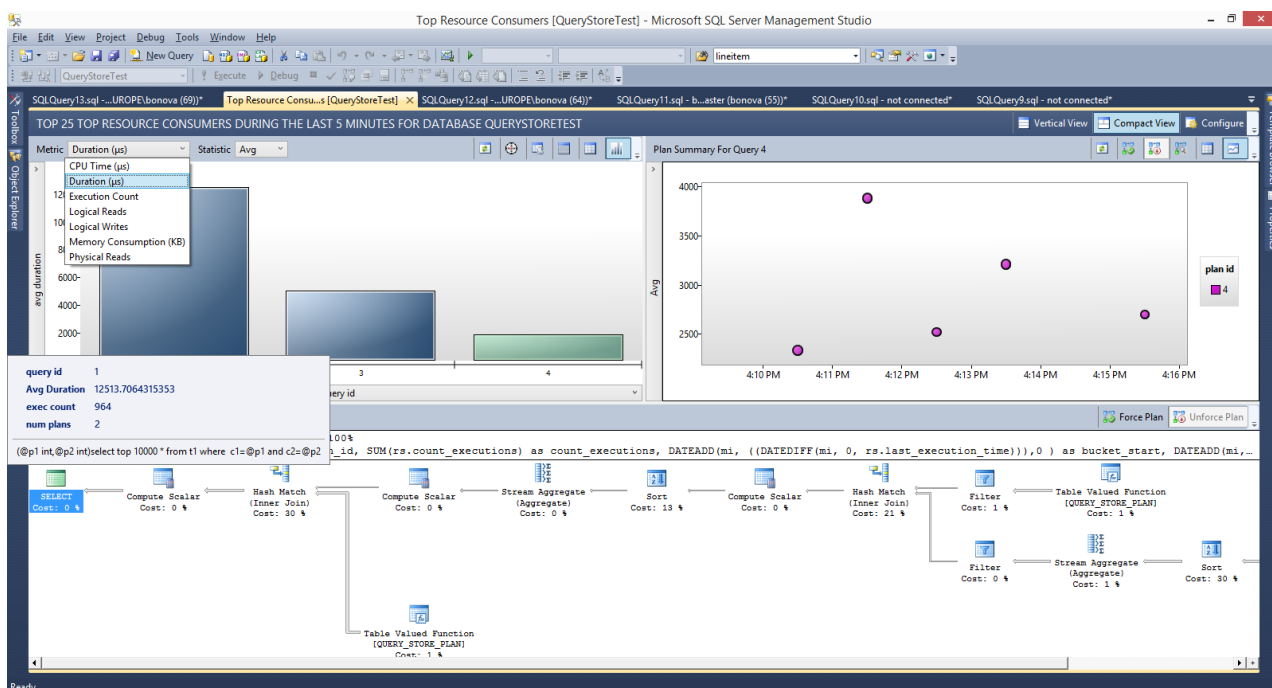


For detailed description of the scenario refer to [Query Store: A flight data recorder for your database](#) blog.

Identify and tune top resource consuming queries

Although your workload may generate thousands of queries, typically only a handful of them actually use the most of the system resources and therefore require your attention. Among top resource consuming queries you will typically find those that are either regressed or those that can be improved with additional tuning.

The easiest way to start exploration is to open **Top Resource Consuming Queries** in Management Studio. User interface is separated into three panes: A histogram representing top resource consuming queries (left), a plan summary for selected query (right) and visual query plan for selected plan (bottom). Click the **Configure** button to control how many queries you want to analyze and the time interval of interest. Additionally, you can choose between different resource consumption dimensions (duration, CPU, memory, IO, number of executions) and the baseline (Average, Min, Max, Total, Standard Deviation).



Look at the plan summary on the right to analyze the execution history and learn about the different plans and

their runtime statistics. Use the bottom pane to examine the different plans or to compare them visually, rendered side by side (use the Compare button).

When you identify a query with sub-optimal performance, your action depends on the nature of the problem:

1. If the query was executed with multiple plans and the last plan is significantly worse than previous plan, you can use the plan forcing mechanism to ensure SQL Server will use the optimal plan for future executions
2. Check if the optimizer is suggesting any missing indexes in XML plan. If yes, create the missing index and use the Query Store to evaluate query performance after the index creation
3. Make sure that the statistics are up-to-date for the underlying tables used by the query.
4. Make sure that indexes used by the query are defragmented.
5. Consider rewriting expensive query. For example, take advantages of query parameterization and reduce usage of dynamic SQL. Implement optimal logic when read the data (apply data filtering on database side, not on application side).

A/B testing

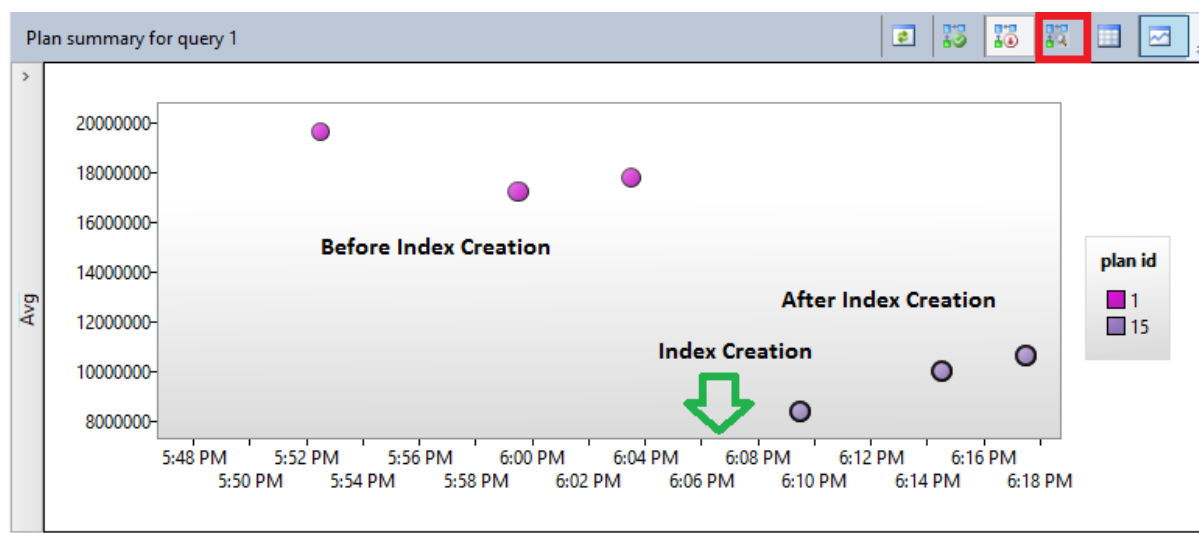
Use Query Store to compare workload performance before and after the application change you plan to introduce. The following list contains several examples where you can use Query Store to assess impact of the environment or application change to the workload performance:

- Rolling out new application version.
- Adding new hardware to the server.
- Creating missing indexes on tables referenced by expensive queries.
- Applying filtering policy for row-level security. For more details see [Optimizing Row Level Security with Query Store](#).
- Adding temporal system-versioning to tables that are frequently modified by your OLTP applications.

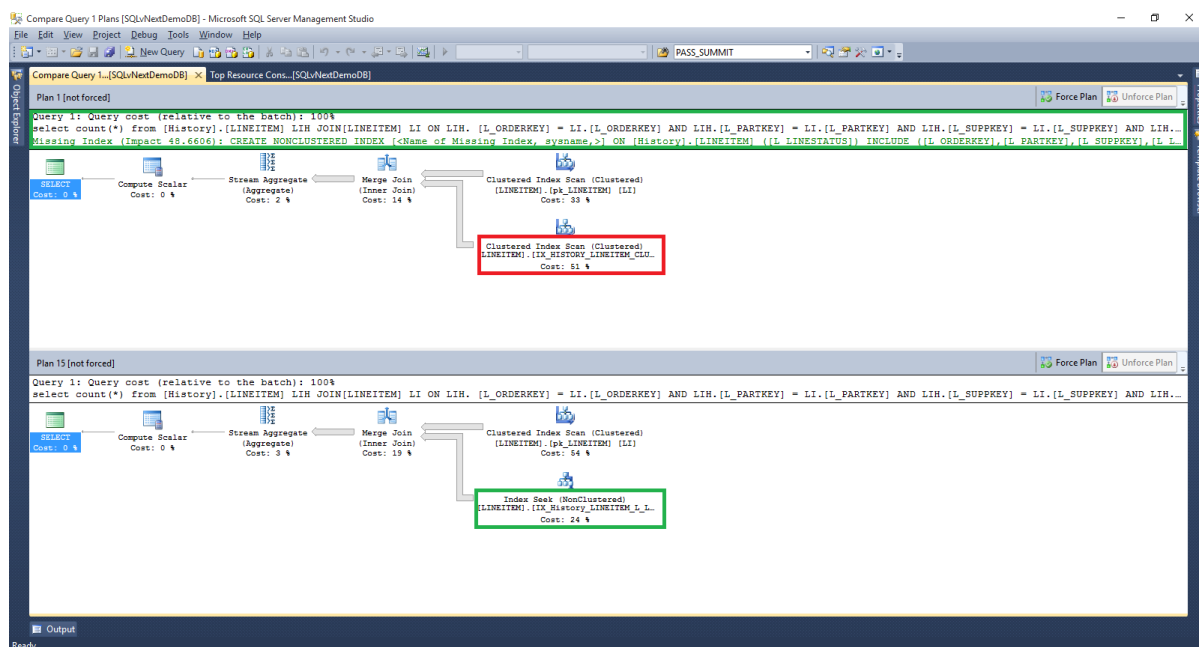
In any of these scenarios apply the following workflow:

1. Run your workload with the Query Store before the planned change to generate performance baseline.
2. Apply application change at the controlled moment in time.
3. Continue running the workload long enough to generate performance image of the system after the change
4. Compare results from #1 and #3.
 - a. Open **Overall Database Consumption** to determine impact to the entire database.
 - b. Open **Top Resource Consuming Queries** (or run your own analysis using Transact-SQL) to analyze impact of the change to the most important queries.
5. Decide whether to keep the change or perform roll back in case when new performance is unacceptable.

The following illustration shows Query Store analysis (step 4) in case of missing index creation. Open **Top Resource Consuming Queries** / Plan summary pane to get this view for the query that should be impacted by the index creation:



Additionally, you can compare plans before and after index creation by rendering them side by side. ("Compare the plans for the selected query in a separate window" toolbar option which is marked with red square on the toolbar.)



Plan before index creation (plan_id = 1, above) has missing index hint and you can inspect that Clustered Index Scan was the most expensive operator in the query (red rectangle).

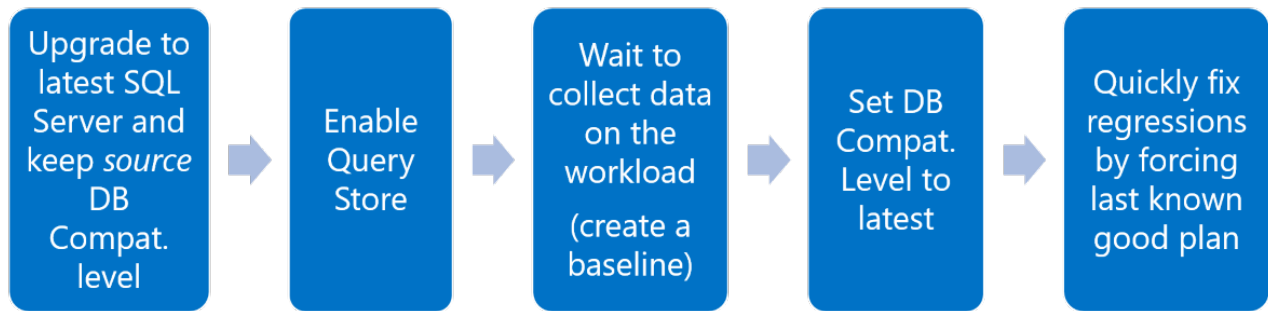
Plan after missing index creation (plan_id = 15, below) now has Index Seek (Nonclustered) which reduces the overall cost of the query and improves its performance (green rectangle).

Based on analysis you would likely keep the index as query performance has been improved.

Keep performance stability during the upgrade to newer SQL Server

Prior to SQL Server 2014, users were exposed to the risk of performance regression during the upgrade to the latest platform version. The reason for that was the fact that the latest version of Query Optimizer became active immediately once new bits are installed.

Starting with SQL Server 2014 all Query Optimizer changes are tied to the latest [database compatibility level](#), so plans are not changed right at point of upgrade but rather when a user changes the `COMPATIBILITY_LEVEL` to the latest one. This capability, in combination with Query Store gives you a great level of control over the query performance in the upgrade process. Recommended upgrade workflow is shown in the following picture:

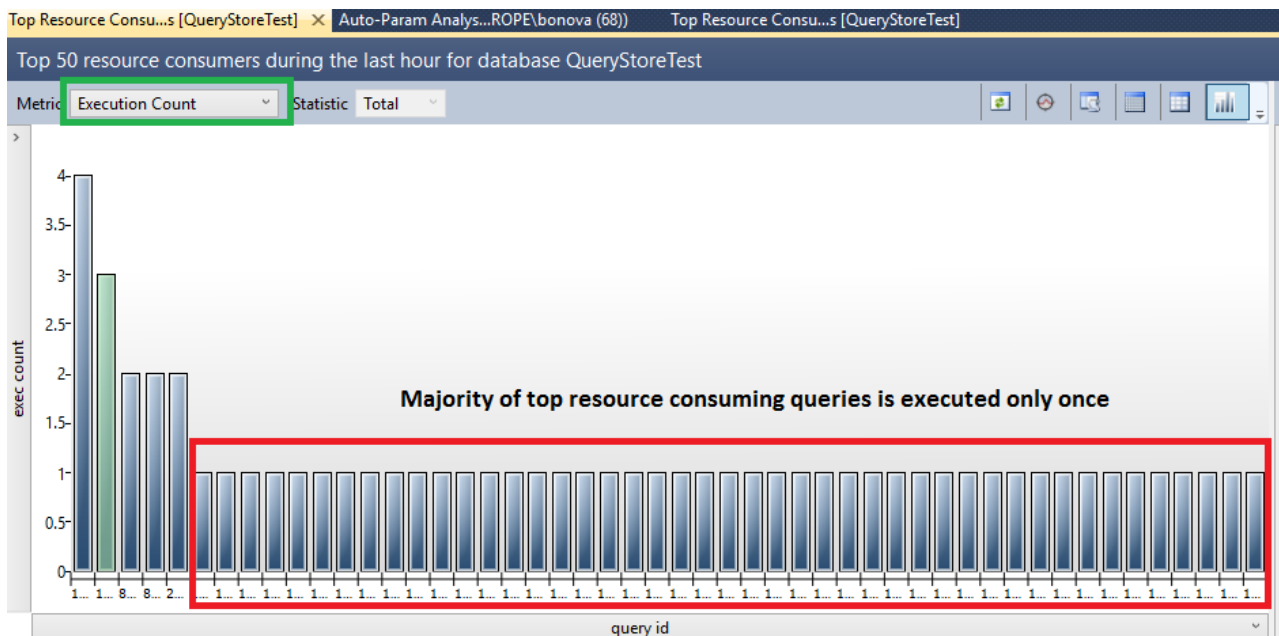


1. Upgrade SQL Server without changing the database compatibility level. It doesn't expose the latest Query Optimizer changes but still provides newer SQL Server features including Query Store.
2. Enable Query Store. For more information on this topic, see [Keep Query Store adjusted to your workload](#).
3. Allow Query Store to capture queries and plans, and establishes a performance baseline with the source/previous database compatibility level. Stay at this step long enough to capture all plans and get a stable baseline. This can be the duration of an usual business cycle for a production workload.
4. Move to latest database compatibility level: get your workload exposed to the latest Query Optimizer changes and let it potentially create new plans.
5. Use Query Store for analysis and regression fixes: for the most part, the new Query Optimizer changes should produce better plans. However, Query Store will provide an easy way to identify plan choice regressions and fix them using a plan forcing mechanism. Starting with SQL Server 2017, when using the [Automatic Plan Correction](#) feature, this step becomes automatic.

Identify and improve ad-hoc workloads

Some workloads do not have dominant queries that you can tune to improve overall application performance. Those workloads are typically characterized with relatively large number of different queries each of them consuming portion of system resources. Being unique, those queries are executed very rarely (usually only once, thus name ad hoc), so their runtime consumption is not critical. On the other hand, given that application is generating net new queries all the time, significant portion of system resources is spent on query compilation which is not optimal. This is not ideal situation for Query Store either given that large number of queries and plans flood the space you have reserved which means that Query Store will likely end up in the read-only mode very quickly. If you activated **Size Based Cleanup Policy** ([highly recommended](#) to keep Query Store always up and running), then background process will be cleaning Query Store structures most of the time also taking significant system resources.

Top Resource Consuming Queries view will give you first indication of the ad-hoc nature of your workload:



Use **Execution Count** metric to analyze whether your top queries are ad hoc (this requires you to run Query Store with `QUERY_CAPTURE_MODE = ALL`). From diagram above you can see that 90% of your **Top Resource Consuming Queries** are executed only once.

Alternatively, you can run Transact-SQL script to get total number of query texts, queries and plans in the system and determine how different they are by comparing their `query_hash` and `plan_hash`:

```
/*Do cardinality analysis when suspect on ad-hoc workloads*/
SELECT COUNT(*) AS CountQueryTextRows FROM sys.query_store_query_text;
SELECT COUNT(*) AS CountQueryRows FROM sys.query_store_query;
SELECT COUNT(DISTINCT query_hash) AS CountDifferentQueryRows FROM sys.query_store_query;
SELECT COUNT(*) AS CountPlanRows FROM sys.query_store_plan;
SELECT COUNT(DISTINCT query_plan_hash) AS CountDifferentPlanRows FROM sys.query_store_plan;
```

This is one potential result you can get in case of workload with ad-hoc queries:

100 %	Results	Messages
CountQueryTextRows		
1	3393	
CountQueryRows		
1	3393	
CountDifferentQueryRows		
1	19	
CountPlanRows		
1	3395	
CountDifferentPlanRows		
1	20	

Query result shows that despite the large number of queries and plans in the Query Store their `query_hash` and `plan_hash` are actually not different. Ratio between unique query texts and unique `query_hash` which is much bigger than 1 is an indication that workload is a good candidate for parameterization as the only difference between the queries is literal constant (parameter) provided as part of the query text.

Usually, this situation happens if your application generates queries (instead of invoking stored procedures or parameterized queries) or if it relies on object-relational mapping frameworks that generate queries by default.

If you are in control of the application code you may consider rewriting of the data access layer to utilize stored procedures or parameterized queries. However, this situation can be also significantly improved without application changes by forcing query parameterization for the entire database (all queries) or for the individual query templates with the same query_hash.

Approach with individual query templates requires plan guide creation:

```
/*Apply plan guide for the selected query template*/
DECLARE @stmt nvarchar(max);
DECLARE @params nvarchar(max);
EXEC sp_get_query_template
    N'<your query text goes here>',
    @stmt OUTPUT,
    @params OUTPUT;

EXEC sp_create_plan_guide
    N'TemplateGuide1',
    @stmt,
    N'TEMPLATE',
    NULL,
    @params,
    N'OPTION (PARAMETERIZATION FORCED)';
```

Solution with plan guides is more precise but it requires more work.

If all your queries (or majority of them) are candidates for auto-parameterization than changing

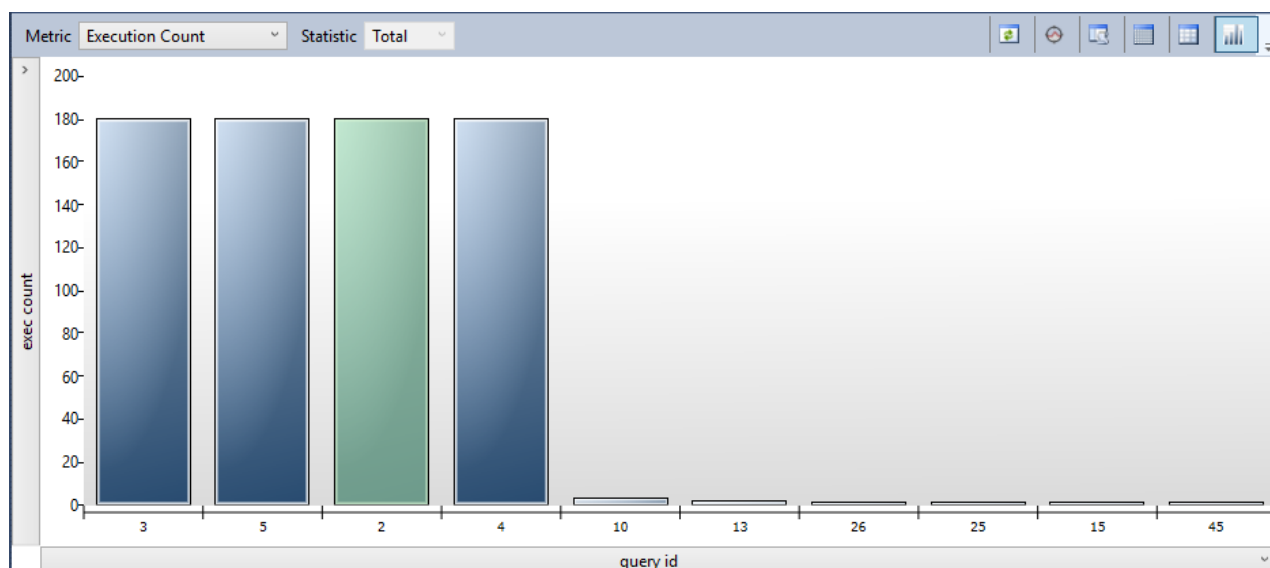
FORCED PARAMETERIZATION for the entire database may be a better option:

```
/*Apply forced parameterization for entire database*/
ALTER DATABASE <database name> SET PARAMETERIZATION FORCED;
```

NOTE

For more information on this topic, see [Guidelines for Using Forced Parameterization](#).

After you apply any of these steps, **Top Resource Consuming Queries** will show you different picture of your workload.



In some cases your application may generate lots of different queries which are not good candidates for auto-parameterization. In that case you will see large number of queries in the system but the ratio between unique

queries and unique `query_hash` is likely close to 1.

In that case you may want to enable the **Optimize for Ad Hoc Workloads** server option to prevent wasting cache memory on queries that won't likely be executed again. To prevent capture of those queries in the Query Store, set `QUERY_CAPTURE_MODE` to `AUTO`.

```
sp_configure 'show advanced options', 1;
GO
RECONFIGURE;
GO

sp_configure 'optimize for ad hoc workloads', 1;
GO
RECONFIGURE;
GO

ALTER DATABASE [QueryStoreTest] SET QUERY_STORE CLEAR;
ALTER DATABASE [QueryStoreTest] SET QUERY_STORE = ON
    (OPERATION_MODE = READ_WRITE, QUERY_CAPTURE_MODE = AUTO);
```





See Also

[Monitoring Performance By Using the Query Store](#)

[Best Practice with the Query Store](#)

How Query Store Collects Data

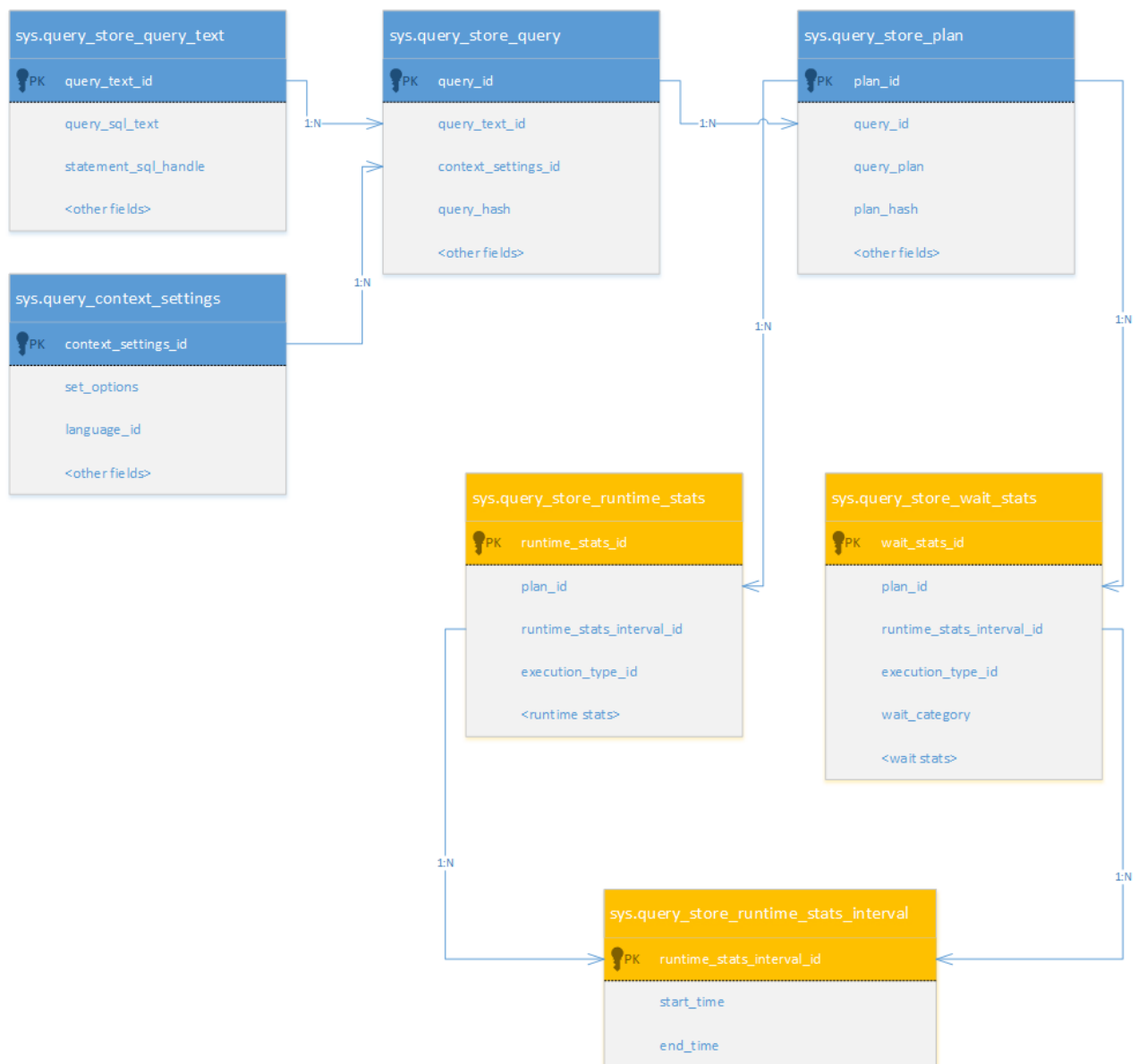
1/17/2018 • 3 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

Query Store works as a **flight data recorder** constantly collecting compile and runtime information related to the queries and plans. Query related data is persisted in the internal tables and presented to users through a set of views.

Views

The following diagram shows Query Store views and their logical relationships, with compile time information presented as blue entities:



View Descriptions

VIEW	DESCRIPTION
sys.query_store_query_text	Presents unique query texts executed against the database. Comments and spaces before and after the query text are ignored. Comments and spaces inside text are not ignored. Every statement in the batch generates a separate query text entry.
sys.query_context_settings	Presents unique combinations of plan affecting settings under which queries are executed. The same query text executed with different plan affecting settings produces separate query entry in the Query Store because <code>context_settings_id</code> is part of the query key.
sys.query_store_query	Query entries that are tracked and forced separately in the Query Store. A single query text can produce multiple query entries if it is executed under different context settings or if it is executed outside vs. inside of different Transact-SQL modules (stored procedures, triggers, etc.).
sys.query_store_plan	Presents estimated plan for the query with the compile time statistics. Stored plan is equivalent to one that you would get by using <code>SET SHOWPLAN_XML ON</code> .
sys.query_store_runtime_stats_interval	Query Store divides time into automatically generated time windows (intervals) and stores aggregated statistics on that interval for every executed plan. The size of the interval is controlled by the configuration option Statistics Collection Interval (in Management Studio) or <code>INTERVAL_LENGTH_MINUTES</code> using ALTER DATABASE SET Options (Transact-SQL) .
sys.query_store_runtime_stats	Aggregated runtime statistics for executed plans. All captured metrics are expressed in form of 4 statistic functions: Average, Minimum, Maximum, and Standard Deviation.

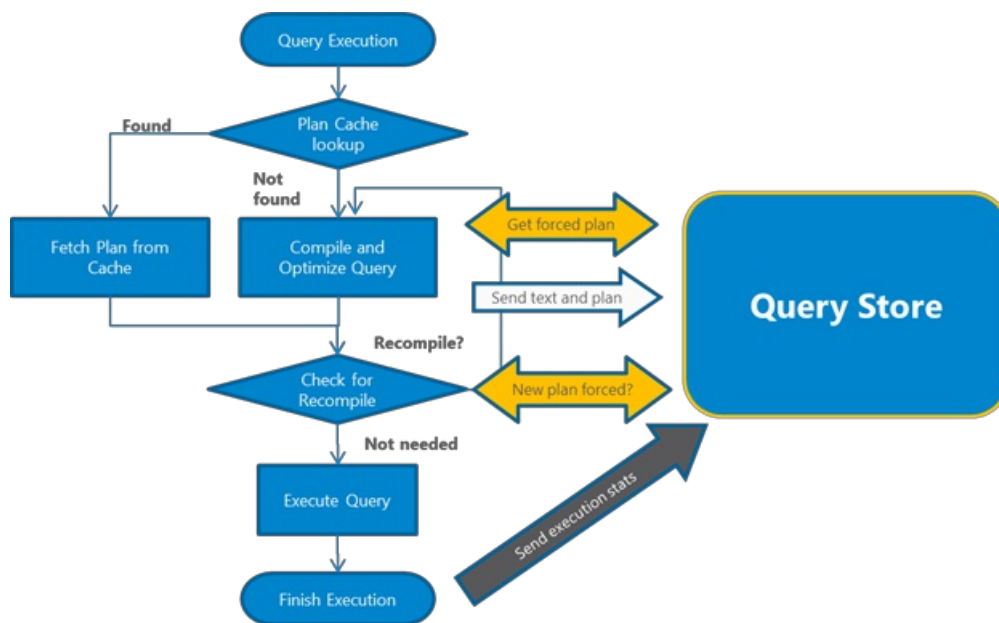
For additional details on Query Store views, see the **Related Views, Functions, and Procedures** section of [Monitoring Performance By Using the Query Store](#).

Query Processing

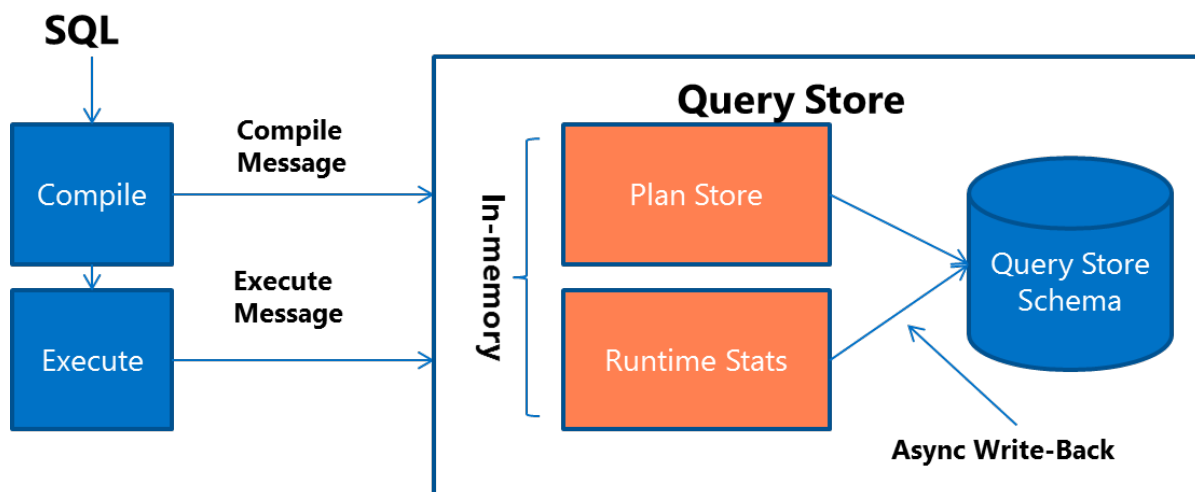
Query store interacts with the query processing pipeline at the following key points:

1. When query gets compiled for the first time, query text and initial plan are sent to Query Store
2. When query gets recompiled, the plan is being updated in the Query Store. If a new plan is created, Query Store adds the new plan entry for the query, keeping the previous ones along with their execution statistics.
3. Upon the query execution, runtime statistics are sent to the Query Store. Query Store keeps aggregated statistics accurate for every plan that was executed within the currently active interval.
4. During the compile and check for recompile phases, SQL Server determines if there is a plan in the Query Store that should be applied for the currently running query. If there is a forced plan and the plan in the procedure cache is different than the forced plan, the query gets recompiled, effectively the same way as if `PLAN HINT` was applied to that query. This process happens transparently to the user application.

The following diagram depicts points of integration explained above:



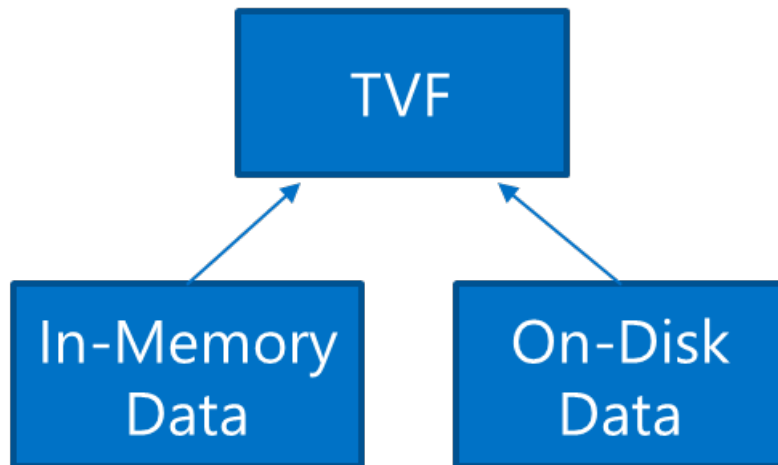
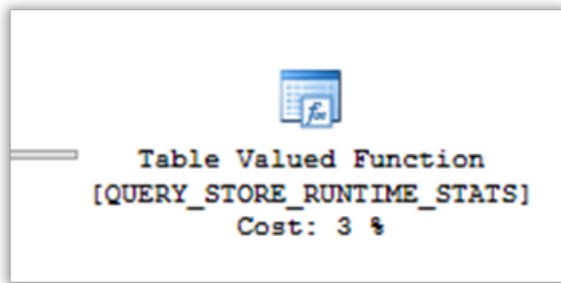
To minimize I/O overhead, new data is captured in-memory. Writes operations are queued and flushed to disk afterwards. Query and plan information (Plan Store on diagram below) are flushed with minimal latency. The runtime statistics (Runtime Stats) are kept in memory for a period of time defined with the `DATA_FLUSH_INTERVAL_SECONDS` option of the `SET QUERY_STORE` statement. The SSMS Query Store dialog box allows you to enter **Data Flush Interval (Minutes)**, which it converts to seconds.



In case of a system crash, Query Store can lose runtime data up to amount defined with `DATA_FLUSH_INTERVAL_SECONDS`. The default value of 900 seconds (15 minutes) is an optimal balance between query capture performance and data availability.

In case of memory pressure, runtime statistics can be flushed to disk earlier than defined with `DATA_FLUSH_INTERVAL_SECONDS`.

During the read of the Query Store data in-memory and on-disk data are unified transparently. In case of session termination or client application restart/crash query statistics won't be recorded.



See Also





[Monitoring Performance By Using the Query Store](#)

[Best Practice with the Query Store](#)

[Query Store Catalog Views \(Transact-SQL\)](#)

Database Engine Tuning Advisor

1/17/2018 • 6 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

The Microsoft Database Engine Tuning Advisor (DTA) analyzes databases and makes recommendations that you can use to optimize query performance. You can use the Database Engine Tuning Advisor to select and create an optimal set of indexes, indexed views, or table partitions without having an expert understanding of the database structure or the internals of SQL Server. Using the DTA, you can perform the following tasks.

- Troubleshoot the performance of a specific problem query
- Tune a large set of queries across one or more databases
- Perform an exploratory what-if analysis of potential physical design changes
- Manage storage space

Database Engine Tuning Advisor Benefits

Optimizing query performance can be difficult without a full understanding the database structure and the queries that are run against the database. The Database Engine Tuning Advisor can make this task easier by analyzing the current query plan cache or by analyzing a workload of Transact-SQL queries that you create and recommending an appropriate physical design. For more advanced database administrators, DTA exposes a powerful mechanism to perform exploratory what-if analysis of different physical design alternatives. The DTA can provide the following information.

- Recommend the best mix of rowstore and [columnstore](#) indexes for databases by using the query optimizer to analyze queries in a workload.
- Recommend aligned or non-aligned partitions for databases referenced in a workload.
- Recommend indexed views for databases referenced in a workload.
- Analyze the effects of the proposed changes, including index usage, query distribution among tables, and query performance in the workload.
- Recommend ways to tune the database for a small set of problem queries.
- Allow you to customize the recommendation by specifying advanced options such as disk space constraints.
- Provide reports that summarize the effects of implementing the recommendations for a given workload.
- Consider alternatives in which you supply possible design choices in the form of hypothetical configurations for Database Engine Tuning Advisor to evaluate.
- Tune workloads from a variety of sources including SQL Server Query Store, Plan Cache, SQL Server Profiler Trace file or table, or a .SQL file.

The Database Engine Tuning Advisor is designed to handle the following types of query workloads.

- Online transaction processing (OLTP) queries only
- Online analytical processing (OLAP) queries only

- Mixed OLTP and OLAP queries
- Query-heavy workloads (more queries than data modifications)
- Update-heavy workloads (more data modifications than queries)

DTA Components and Concepts

Database Engine Tuning Advisor Graphical User Interface

An easy-to-use interface in which you can specify the workload and select various tuning options.

dta Utility

The command prompt version of Database Engine Tuning Advisor. The **dta** utility is designed to allow you to use Database Engine Tuning Advisor functionality in applications and scripts.

workload

A Transact-SQL script file, trace file, or trace table that contains a representative workload for the databases you want to tune. Beginning with SQL Server 2012, you can specify the plan cache as the workload. Beginning with SQL Server 2016, you can [specify the Query Store as the workload](#).

XML input file

An XML-formatted file that Database Engine Tuning Advisor can use to tune workloads. The XML input file supports advanced tuning options that are not available in either the GUI or **dta** utility.

Limitations and Restrictions

The Database Engine Tuning Advisor has the following limitations and restrictions.

- It cannot add or drop unique indexes or indexes that enforce PRIMARY KEY or UNIQUE constraints.
- It cannot analyze a database that is set to single-user mode.
- If you specify a maximum disk space for tuning recommendations that exceeds the actual available space, Database Engine Tuning Advisor uses the value you specify. However, when you execute the recommendation script to implement it, the script may fail if more disk space is not added first. Maximum disk space can be specified with the **-B** option of the **dta** utility, or by entering a value in the **Advanced Tuning Options** dialog box.
- For security reasons, Database Engine Tuning Advisor cannot tune a workload in a trace table that resides on a remote server. To work around this limitation, you can use a trace file instead of a trace table or copy the trace table to the remote server.
- When you impose constraints, such as those imposed when you specify a maximum disk space for tuning recommendations (by using the **-B** option or the **Advanced Tuning Options** dialog box), Database Engine Tuning Advisor may be forced to drop certain existing indexes. In this case, the resulting Database Engine Tuning Advisor recommendation may produce a negative expected improvement.
- When you specify a constraint to limit tuning time (by using the **-A** option with the **dta** utility or by checking **Limit tuning time** on the **Tuning Options** tab), Database Engine Tuning Advisor may exceed that time limit to produce an accurate expected improvement and the analysis reports for whatever portion of the workload has been consumed so far.
- Database Engine Tuning Advisor might not make recommendations under the following circumstances:
 1. The table being tuned contains less than 10 data pages.
 2. The recommended indexes would not offer enough improvement in query performance over the current physical database design.

3. The user who runs Database Engine Tuning Advisor is not a member of the **db_owner** database role or the **sysadmin** fixed server role. The queries in the workload are analyzed in the security context of the user who runs the Database Engine Tuning Advisor. The user must be a member of the **db_owner** database role.
- Database Engine Tuning Advisor stores tuning session data and other information in the **msdb** database. If changes are made to the **msdb** database you may risk losing tuning session data. To eliminate this risk, implement an appropriate backup strategy for the **msdb** database.

Performance Considerations

Database Engine Tuning Advisor can consume significant processor and memory resources during analysis. To avoid slowing down your production server, follow one of these strategies:

- Tune your databases when your server is free. Database Engine Tuning Advisor can affect maintenance task performance.
- Use the test server/production server feature. For more information, see [Reduce the Production Server Tuning Load](#).
- Specify only the physical database design structures you want Database Engine Tuning Advisor to analyze. Database Engine Tuning Advisor provides many options, but specifies only those that are necessary.

Dependency on xp_msver Extended Stored Procedure

Database Engine Tuning Advisor depends on the **xp_msver** extended stored procedure to provide full functionality. This extended stored procedure is turned on by default. Database Engine Tuning Advisor uses this extended stored procedure to fetch the number of processors and available memory on the computer where the database that you are tuning resides. If **xp_msver** is unavailable, Database Engine Tuning Advisor assumes the hardware characteristics of the computer where Database Engine Tuning Advisor is running. If the hardware characteristics of the computer where Database Engine Tuning Advisor is running are not available, one processor and 1024 megabytes (MBs) of memory are assumed.

This dependency affects partitioning recommendations because the number of partitions recommended depends on these two values (number of processors and available memory). The dependency also affects your tuning results when you use a test server to tune your production server. In this scenario, Database Engine Tuning Advisor uses **xp_msver** to fetch hardware properties from the production server. After tuning the workload on the test server, Database Engine Tuning Advisor uses these hardware properties to generate a recommendation. For more information, see [xp_msver \(Transact-SQL\)](#).





Database Engine Tuning Advisor Tasks

The following table lists common Database Engine Tuning Advisor tasks and the topics that describe how to perform them.

DATABASE ENGINE TUNING ADVISOR TASK	TOPIC
<p>Initialize and start the Database Engine Tuning Advisor.</p> <p>Create a workload by specifying the plan cache, by creating a script, or by generating a trace file or trace table.</p> <p>Tune a database by using the Database Engine Tuning Advisor graphical user interface tool.</p> <p>Create XML input files to tune workloads.</p> <p>View descriptions of the Database Engine Tuning Advisor user interface options.</p>	<p>Start and Use the Database Engine Tuning Advisor</p>
<p>View the results of the database tuning operation.</p> <p>Select and implement tuning recommendations.</p> <p>Perform what-if exploratory analysis against the workload.</p> <p>Review existing tuning sessions, clone sessions based on existing ones or edit existing tuning recommendations for further evaluation or implementation.</p> <p>View descriptions of the Database Engine Tuning Advisor user interface options.</p>	<p>View and Work with the Output from the Database Engine Tuning Advisor</p>

Start and Use the Database Engine Tuning Advisor

1/17/2018 • 24 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

This topic describes how to start and use Database Engine Tuning Advisor in SQL Server 2017. For information about how to view and work with the results after you tune a database, see [View and Work with the Output from the Database Engine Tuning Advisor](#).

Initialize the Database Engine Tuning Advisor

On first use, a user who is member of the **sysadmin** fixed server role must initialize the Database Engine Tuning Advisor. This is because several system tables must be created in the **msdb** database to support tuning operations. Initialization also enables users that are members of the **db_owner** fixed database role to tune workloads on tables in databases that they own.

A user that has system administrator permissions must perform either of the following actions:

- Use the Database Engine Tuning Advisor graphical user interface to connect to an instance of SQL Server 2017. For more information, see [Start the Database Engine Tuning Advisor](#) later in this topic.
- Use the **dta** utility to tune the first workload. For more information, see [Use the dta Utility](#) later in this topic.

Start the Database Engine Tuning Advisor

You can start the Database Engine Tuning Advisor graphical user interface (GUI) in several different ways to support database tuning in a variety of scenarios. The different ways to start Database Engine Tuning Advisor include: from the **Start** menu, from the **Tools** menu in SQL Server Management Studio, from the Query Editor in SQL Server Management Studio, and from the **Tools** menu in SQL Server Profiler. When you first start Database Engine Tuning Advisor, the application displays a **Connect to Server** dialog box where you can specify the SQL Server instance to which you want to connect.

WARNING

Do not start Database Engine Tuning Advisor when SQL Server is running in single-user mode. If you attempt to start it while the server is in single-user mode, an error will be returned and Database Engine Tuning Advisor will not start. For more information about single-user mode, see [Start SQL Server in Single-User Mode](#).

To start Database Engine Tuning Advisor from the Windows Start menu

1. On the **Start** menu, point to **All Programs**, point to **Microsoft SQL Server**, point to **Performance Tools**, and then click **Database Engine Tuning Advisor**.

To start the Database Engine Tuning Advisor in SQL Server Management Studio

1. On the SQL Server Management Studio **Tools** menu, click **Database Engine Tuning Advisor**.

To start the Database Engine Tuning Advisor from the SQL Server Management Studio Query Editor

1. Open a Transact-SQL script file in SQL Server Management Studio. For more information, see [Query and Text Editors \(SQL Server Management Studio\)](#).
2. Select a query in the Transact-SQL script, or select the entire script, right-click the selection, and choose

Analyze Query in Database Engine Tuning Advisor. The Database Engine Tuning Advisor GUI opens and imports the script as an XML file workload. You can specify a session name and tuning options to tune the selected Transact-SQL queries as your workload.

To start the Database Engine Tuning Advisor in SQL Server Profiler

1. On the SQL Server Profiler **Tools** menu, click **Database Engine Tuning Advisor**.

Create a Workload

A workload is a set of Transact-SQL statements that execute against a database or databases that you want to tune. Database Engine Tuning Advisor analyzes these workloads to recommend indexes or partitioning strategies that will improve your server's query performance.

You can create a workload by using one of the following methods.

- Use the [Query Store](#) as a workload. By doing this, you can avoid having to manually create a workload. For more information, see [Tuning Database Using Workload From Query Store](#).

|Applies to: SQL Server 2016 through SQL Server 2017.]

- Use the plan cache as a workload. By doing this, you can avoid having to manually create a workload. For more information, see [Tune a Database](#) later in this topic.
- Use the Query Editor in SQL Server Management Studio or your favorite text editor to manually create Transact-SQL script workloads.
- Use SQL Server Profiler to create trace file or trace table workloads

NOTE

When using a trace table as a workload, that table must exist on the same server where Database Engine Tuning Advisor is tuning. If you create the trace table on a different server, then move it to the server where Database Engine Tuning Advisor is tuning.

- Workloads can also be embedded in an XML input file, where you can also specify a weight for each event. For more information about specifying embedded workloads, see [Create an XML Input File](#) later in this topic.

To create Transact-SQL script workloads

1. Launch the Query Editor in SQL Server Management Studio. For more information, see [Query and Text Editors \(SQL Server Management Studio\)](#).
2. Type your Transact-SQL script into the Query Editor. This script should contain a set of Transact-SQL statements that execute against the database or databases that you want to tune.
3. Save the file with a **.sql** extension. The Database Engine Tuning Advisor GUI and the command-line **dta** utility can use this Transact-SQL script as a workload.

To create trace file and trace table workloads

1. Launch SQL Server Profiler by using one of the following methods:
 - On the **Start** menu, point to **All Programs, Microsoft SQL Server, Performance Tools**, and then click **SQL Server Profiler**.
 - In SQL Server Management Studio, click the **Tools** menu, and then click **SQL Server Profiler**.

2. Create a trace file or table as described in the following procedures that uses the SQL Server Profiler **Tuning** template:

- [Create a Trace \(SQL Server Profiler\)](#)
- [Save Trace Results to a File \(SQL Server Profiler\)](#)

Database Engine Tuning Advisor assumes that the workload trace file is a rollover file. For more information about rollover files, see [Limit Trace File and Table Sizes](#).

- [Save Trace Results to a Table \(SQL Server Profiler\)](#)

Make sure that tracing has stopped before using a trace table as a workload.

We recommend that you use the SQL Server Profiler Tuning template for capturing workloads for Database Engine Tuning Advisor.

If you want to use your own template, ensure that the following trace events are captured:

- **RPC:Completed**
- **SQL:BatchCompleted**
- **SP:StmtCompleted**

You can also use the **Starting** versions of these trace events. For example, **SQL:BatchStarting**. However, the **Completed** versions of these trace events include the **Duration** column, which allows Database Engine Tuning Advisor to more effectively tune the workload. Database Engine Tuning Advisor does not tune other types of trace events. For more information about these trace events, see [Stored Procedures Event Category](#) and [TSQL Event Category](#). For information about using the SQL Trace stored procedures to create a trace file workload, see [Create a Trace \(Transact-SQL\)](#).

Trace File or Trace Table Workloads That Contain the LoginName Data Column

Database Engine Tuning Advisor submits Showplan requests as part of the tuning process. When a trace table or file that contains the **LoginName** data column is consumed as a workload, Database Engine Tuning Advisor impersonates the user specified in **LoginName**. If this user has not been granted the SHOWPLAN permission, which enables the user to execute and produce Showplans for the statements contained in the trace, Database Engine Tuning Advisor will not tune those statements.

To avoid granting the SHOWPLAN permission to each user specified in the LoginName column of the trace

1. Tune the trace file or table workload. For more information, see [Tune a Database](#) later in this topic.
2. Check the tuning log for statements that were not tuned due to inadequate permissions. For more information, see [View and Work with the Output from the Database Engine Tuning Advisor](#).
3. Create a new workload by deleting the **LoginName** column from the events that were not tuned, and then save only the untuned events in a new trace file or table. For more information about deleting data columns from a trace, see [Specify Events and Data Columns for a Trace File \(SQL Server Profiler\)](#) or [Modify an Existing Trace \(Transact-SQL\)](#).
4. Resubmit the new workload without the **LoginName** column to Database Engine Tuning Advisor.

Database Engine Tuning Advisor will tune the new workload because login information is not specified in the trace. If the **LoginName** does not exist for a statement, Database Engine Tuning Advisor tunes that statement by impersonating the user who started the tuning session (a member of either the **sysadmin** fixed server role or the **db_owner** fixed database role).

Tune a Database

To tune a database, you can use the Database Engine Tuning Advisor GUI or the **dta** utility.

NOTE

Make sure that tracing has stopped before using a trace table as a workload for Database Engine Tuning Advisor. Database Engine Tuning Advisor does not support using a trace table to which trace events are still being written as a workload.

Use the Database Engine Tuning Advisor Graphical User Interface

On the Database Engine Tuning Advisor GUI, you can tune a database by using the plan cache, workload files, or workload tables. You can use the Database Engine Tuning Advisor GUI to easily view the results of your current tuning session and results of previous tuning sessions. For information about user interface options, see [User Interface Descriptions](#) later in this topic. For more information about working with the output after you tune a database, see [View and Work with the Output from the Database Engine Tuning Advisor](#).

To tune a database by using the Query Store

See [Tuning Database Using Workload from Query Store](#) for more information.

To tune a database by using the plan cache

1. Launch Database Engine Tuning Advisor, and log into an instance of SQL Server. For more information, see [Start the Database Engine Tuning Advisor](#) earlier in this topic.
2. On the **General** tab, type a name in **Session name** to create a new tuning session. You must configure the fields in the **General** tab before starting a tuning session. It is not necessary to modify the settings of the **Tuning Options** tab before starting a tuning session.
3. Select **Plan Cache** as the workload option. Database Engine Tuning Advisor selects the top 1,000 events from the plan cache to use for analysis.
4. Select the database or databases that you want to tune, and optionally from **Selected Tables**, choose one or more tables from each database. To include cache entries for all databases, from **Tuning Options**, click **Advanced Options** and then check **Include plan cache events from all databases**.
5. Check **Save tuning log** to save a copy of the tuning log. Clear the check box if you do not want to save a copy of the tuning log.

You can view the tuning log after analysis by opening the session and selecting the **Progress** tab.

6. Click the **Tuning Options** tab and select from the options listed there.
7. Click **Start Analysis**.

If you want to stop the tuning session after it has started, choose one of the following options on the **Actions** menu:

- **Stop Analysis (With Recommendations)** stops the tuning session and prompts you to decide whether you want Database Engine Tuning Advisor to generate recommendations based on the analysis done up to this point.
- **Stop Analysis** stops the tuning session without generating any recommendations.

NOTE

Pausing Database Engine Tuning Advisor is not supported. If you click the **Start Analysis** toolbar button after clicking either the **Stop Analysis** or **Stop Analysis (With Recommendations)** toolbar buttons, Database Engine Tuning Advisor starts a new tuning session.

To tune a database using a workload file or table as input

1. Determine the database features (indexes, indexed views, partitioning) you want Database Engine Tuning

Advisor to consider adding, removing, or retaining during analysis.

2. Create a workload. For more information, see [Create a Workload](#) earlier in this topic.
3. Launch Database Engine Tuning Advisor, and log into an instance of Microsoft SQL Server. For more information, see [Start the Database Engine Tuning Advisor](#) earlier in this topic.
4. On the **General** tab, type a name in **Session name** to create a new tuning session.
5. Choose either a **Workload File** or **Table** and type either the path to the file, or the name of the table in the adjacent text box.

The format for specifying a table is

```
database_name.schema_name.table_name
```

To search for a workload file or table, click **Browse**. Database Engine Tuning Advisor assumes that workload files are rollover files. For more information about rollover files, see [Limit Trace File and Table Sizes](#).

When using a trace table as a workload, that table must exist on the same server that Database Engine Tuning Advisor is tuning. If you create the trace table on a different server, move it to the server that Database Engine Tuning Advisor is tuning before using it as your workload.

6. Select the databases and tables against which you wish to run the workload that you selected in step 5. To select the tables, click the **Selected Tables** arrow.
7. Check **Save tuning log** to save a copy of the tuning log. Clear the check box if you do not want to save a copy of the tuning log.

You can view the tuning log after analysis by opening the session and selecting the **Progress** tab.

8. Click the **Tuning Options** tab and select from the options listed there.
9. Click the **Start Analysis** button in the toolbar.

If you want to stop the tuning session after it has started, choose one of the following options on the **Actions** menu:

- **Stop Analysis (With Recommendations)** stops the tuning session and prompts you to decide whether you want Database Engine Tuning Advisor to generate recommendations based on the analysis done up to this point.
- **Stop Analysis** stops the tuning session without generating any recommendations.

NOTE

Pausing Database Engine Tuning Advisor is not supported. If you click the **Start Analysis** toolbar button after clicking either the **Stop Analysis** or **Stop Analysis (With Recommendations)** toolbar buttons, Database Engine Tuning Advisor starts a new tuning session.

Use the dta Utility

The [dta utility](#) provides a command prompt executable file that you can use to tune databases. It enables you to use Database Engine Tuning Advisor functionality in batch files and scripts. The **dta** utility takes plan cache entries, trace files, trace tables, and Transact-SQL scripts as workloads. It also takes XML input that conforms to the Database Engine Tuning Advisor XML schema, which is available at this [Microsoft Web site](#).

Consider the following before you begin tuning a workload with the **dta** utility:

- When using a trace table as a workload, that table must exist on the same server that Database Engine Tuning Advisor is tuning. If you create the trace table on a different server, then move it to the server that Database Engine Tuning Advisor is tuning.
- Make sure that tracing has stopped before using a trace table as a workload for Database Engine Tuning Advisor. Database Engine Tuning Advisor does not support using a trace table to which trace events are still being written as a workload.
- If a tuning session continues running longer than you had anticipated it would run, you can press CTRL+C to stop the tuning session and generate recommendations based on the analysis **dta** has completed up to this point. You will be prompted to decide whether you want to generate recommendations or not. Press CTRL+C again to stop the tuning session without generating recommendations.

For more information about **dta** utility syntax and examples, see [dta Utility](#).

To tune a database by using the plan cache

1. Specify the **-ip** option. The top 1,000 plan cache events for the selected databases are analyzed.

From a command prompt, enter the following:

```
dta -E -D DatabaseName -ip -s SessionName
```

2. To modify the number of events to use for analysis, specify the **-n** option. The following example increases the number of cache entries to 2,000.

```
dta -E -D DatabaseName -ip -n 2000 -s SessionName1
```

3. To analyze events for all databases in the instance, specify the **-ipf** option.

```
dta -E -D DatabaseName -ip -ipf -n 2000 -s SessionName2
```

To tune a database by using a workload and dta utility default settings

1. Determine the database features (indexes, indexed views, partitioning) you want Database Engine Tuning Advisor to consider adding, removing, or retaining during analysis.
2. Create a workload. For more information, see [Create a Workload](#) earlier in this topic.
3. From a command prompt, enter the following:

```
dta -E -D DatabaseName -if WorkloadFile -s SessionName
```

where **-E** specifies that your tuning session uses a trusted connection (instead of a login ID and password), **-D** specifies the name of the database you want to tune. By default, the utility connects to the default instance of SQL Server on the local computer. (Use the **-s** option to specify a remote database as shown in the following procedure, or to specify a named instance.) The **-if** option specifies the name and path to a workload file (which can be a Transact-SQL script or a trace file), and **-s** specifies a name for your tuning session.

The four options shown here (database name, workload, connection type, and session name) are mandatory.

To tune a remote database or a named instance for a specific duration

1. Determine the database features (indexes, indexed views, partitioning) you want Database Engine Tuning

Advisor to consider adding, removing, or retaining during analysis.

2. Create a workload. For more information, see [Create a Workload](#) earlier in this topic.
3. From a command prompt, enter the following:

```
dta -S ServerName\Instance -D DatabaseName -it WorkloadTableName  
-U LoginID -P Password -s SessionName -A TuningTimeInMinutes
```

where `-S` specifies a remote server name and instance (or a named instance on the local server) and `-D` specifies the name of the database you want to tune. The `-it` option specifies the name of the workload table, `-U` and `-P` specify the login ID and password to the remote database, `-s` specifies the tuning session name, and `-A` specifies the tuning session duration in minutes. By default, the **dta** utility uses an 8-hour tuning duration. If you would like Database Engine Tuning Advisor to tune a workload for an unlimited amount of time, specify **0** (zero) with the `-A` option.

To tune a database using an XML input file

1. Determine the database features (indexes, indexed views, partitioning) you want Database Engine Tuning Advisor to consider adding, removing, or retaining during analysis.
2. Create a workload. For more information, see [Create a Workload](#) earlier in this topic.
3. Create an XML input file. For more information, see [Create XML Input Files](#) later in this topic.
4. From a command prompt, enter the following:

```
dta -E -S ServerName\Instance -s SessionName -ix PathToXMLInputFile
```

where `-E` specifies a trusted connection, `-S` specifies a remote server and instance, or a named instance on the local server, `-s` specifies a tuning session name, and `-ix` specifies the XML input file to use for the tuning session.

5. After the utility finishes tuning the workload, you can view the results of tuning sessions with the Database Engine Tuning Advisor GUI. As an alternative, you can also specify that the tuning recommendations be written to an XML file with the `-ox` option. For more information, see [dta Utility](#).

Create an XML Input File

If you are an experienced XML developer, you can create XML-formatted files that Database Engine Tuning Advisor can use to tune workloads. To create these XML files, use your favorite XML tools to edit a sample file or to generate an instance from the Database Engine Tuning Advisor XML schema.

The Database Engine Tuning Advisor XML schema is available in your Microsoft SQL Server installation in the following location:

C:\Program Files\Microsoft SQL Server\100\Tools\Binn\schemas\sqlserver\2004\07\dta\dtaschema.xsd

The Database Engine Tuning Advisor XML schema is also available online at this [Microsoft Web site](#).

This URL takes you to a page where many SQL Server XML schemas are available. Scroll down the page until you reach the row for Database Engine Tuning Advisor.

To create an XML input file to tune workloads

1. Create a workload. You can use a trace file or table by using the tuning template in SQL Server Profiler, or create a Transact-SQL script that reproduces a representative workload for SQL Server. For more information, see [Create a Workload](#) earlier in this topic.

2. Create an XML input file by one of the following methods:

- Copy and paste one of the [XML Input File Samples \(DTA\)](#) into your favorite XML editor. Change the values to specify the appropriate arguments for your SQL Server installation, and save the XML file.
- Using your favorite XML tool, generate an instance from the Database Engine Tuning Advisor XML schema.

3. After creating the XML input file, use it as input to the **dta** command-line utility to tune the workload. For information about using XML input files with this utility, see the section [Use the dta Utility](#) earlier in this topic.

NOTE

If you want to use an inline workload, which is a workload that is specified directly in the XML input file, use the sample [XML Input File Sample with Inline Workload \(DTA\)](#).

User Interface Descriptions

Tools Menu/Options Page

Use this dialog box to specify general configuration parameters for the Database Engine Tuning Advisor.

On startup

Specify what Database Engine Tuning Advisor should do when it is started: open without a database connection, show a **New Connection** dialog box, show a new session, or load the last loaded session.

Change font

Specify the display font used by Database Engine Tuning Advisor tables.

Number of items in most recently used lists

Specify the number of sessions or files to display under **Recent Sessions** or **Recent Files** in the **File** menu.

Remember my last tuning options

Retain tuning options between sessions. Selected by default. Clear this check box to always start with the Database Engine Tuning Advisor defaults.

Ask before permanently deleting sessions

Display a confirmation dialog box before deleting sessions.

Ask before stopping session analysis

Display a confirmation dialog box before stopping analysis of a workload.

General Tab Options

You must configure the fields in the **General** tab before starting a tuning session. You do not have to modify the settings of the **Tuning Options** tab before starting a tuning session.

Session name

Specify a name for the session. The session name associates a name with a tuning session. You can refer to this name to review the tuning session later.

File

Specify a .sql script or trace file for a workload. Specify the path and filename in the associated text box. Database Engine Tuning Advisor assumes that the workload trace file is a rollover file. For more information about rollover files, see [Limit Trace File and Table Sizes](#).

Table

Specify a trace table for a workload. Specify the fully qualified name of the trace table in the associated text box

as follows:

```
database_name.owner_name.table_name
```

- Make sure that tracing has stopped before using a trace table as a workload.
- The trace table must exist on the same server that Database Engine Tuning Advisor is tuning. If you create the trace table on a different server, then move it to the server that Database Engine Tuning Advisor is tuning.

Plan Cache

Specify the plan cache as a workload. By doing this, you can avoid having to manually create a workload. Database Engine Tuning Advisor selects the top 1,000 events to use for analysis.

Xml

This does not appear unless you import a workload query from SQL Server Management Studio.

To import a workload query from SQL Server Management Studio:

1. Type a query into Query Editor and highlight it.
2. Right-click the highlighted query and click **Analyze Query in Database Engine Tuning Advisor**.

Browse for a workload [file or table]

When **File** or **Table** is selected as the workload source, use this browse button to select the target.

Preview the XML workload

View an XML-formatted workload that has been imported from SQL Server Management Studio.

Database for workload analysis

Specify the first database to which Database Engine Tuning Advisor connects when tuning a workload. After tuning begins, Database Engine Tuning Advisor connects to the databases specified by the `USE DATABASE` statements contained in the workload.

Select databases and tables to tune

Specify the databases and tables to be tuned. To specify all of the databases, select the check box in the **Name** column heading. To specify certain databases, select the check box next to the database name. By default, all of the tables for selected databases are automatically included in the tuning session. To exclude tables, click the arrow in the **Selected Tables** column, and then clear the check boxes next to the tables that you do not want to tune.

Selected Tables down arrow

Expand the tables list to allow selecting individual tables for tuning.

Save tuning log

Create a log and record errors during the session.

NOTE

Database Engine Tuning Advisor does not automatically update the rows information for the tables displayed on the **General** tab. Instead it relies upon the metadata in the database. If you suspect that the rows information is outdated, run the DBCC UPDATEUSAGE command for the relevant objects.

Tuning Tab Options

Use the **Tuning Options** tab to modify default settings of general tuning options. You do not have to modify the settings of the **Tuning Options** tab before starting a tuning session.

Limit tuning time

Limits the time for the current tuning session. Providing more time for turning improves the quality of the recommendations. To ensure the best recommendations, do not select this option.

NOTE

Database Engine Tuning Advisor consumes system resources during analysis. Use **Limit tuning time** to stop tuning before periods of anticipated heavy workload on the server being tuned.

Advanced Options

Use the **Advanced Tuning Options** dialog box to configure the maximum space, maximum key columns, and online index recommendations.

Define max. space for recommendations (MB)

Type the maximum amount of space to be used by physical design structures recommended by Database Engine Tuning Advisor.

If no value is entered here, Database Engine Tuning Advisor assumes the smaller of the following space limits:

- Three times the current raw data size, which includes the total size of heaps and clustered indexes on tables in the database.
- The free space on all attached disk drives plus the raw data size.

Include plan cache events from all databases

Specify that plan cache events from all databases are analyzed.

Max. columns per index

Specify the maximum number of columns to include in any index. The default is 1023.

All recommendations are offline

Generate the best recommendations possible, but do not recommend that any physical design structures be created online.

Generate online recommendations where possible

When creating Transact-SQL statements to implement the recommendations, choose methods that can be implemented with the server online, even if a faster offline method is available.

Generate only online recommendations

Only make recommendations that allow the server to stay online.

Stop at

Provide the date and time when Database Engine Tuning Advisor should stop.

Indexes and indexed views

Check this box to include recommendations for adding clustered indexes, nonclustered indexes, and indexed views.

Indexed views

Only include recommendations for adding indexed views. Clustered and nonclustered indexes will not be recommended.

Include filtered indexes

Include recommendations for adding filtered indexes. This option is available if you select one of these physical design structures: **Indexes and indexed views**, **Indexes**, or **Nonclustered indexes**.

Indexes

Only include recommendations for adding clustered and nonclustered indexes. Indexed views will not be

recommended.

Nonclustered indexes

Include recommendations for only nonclustered indexes. Clustered indexes and indexed views will not be recommended.

Evaluate utilization of existing PDS only

Evaluate the effectiveness of the current indexes but do not recommend additional indexes or indexed views.

No partitioning

Do not recommend partitioning.

Full partitioning

Include recommendations for partitioning.

Aligned partitioning

New recommended partitions will be aligned to make partitions easy to maintain.

Do not keep any existing PDS

Recommend dropping unnecessary existing indexes, views, and partitioning. If an existing physical design structure (PDS) is useful to the workload, Database Engine Tuning Advisor does not recommend dropping it.

Keep indexes only

Keep all existing indexes but recommend dropping unnecessary indexed views, and partitioning.

Keep all existing PDS

Keep all existing indexes, indexed views, and partitioning.

Keep clustered indexes only

Keep all existing clustered indexes but recommend dropping unnecessary indexed views, partitions, and nonclustered indexes.

Keep aligned partitioning

Keep partitioning structures that are currently aligned, but recommend dropping unnecessary indexed views, indexes, and non-aligned partitioning. Any additional partitioning recommended will align with the current partitioning scheme.

Progress Tab Options

The **Progress** tab of Database Engine Tuning Advisor appears after Database Engine Tuning Advisor begins analyzing a workload.

If you want to stop the tuning session after it has started, choose one of the following options on the **Actions** menu:

- **Stop Analysis (With Recommendations)** stops the tuning session and prompts you to decide whether you want Database Engine Tuning Advisor to generate recommendations based on the analysis done up to this point.
- **Stop Analysis** stops the tuning session without generating any recommendations.

Tuning Progress

Indicates the current status of the progress. Contains the number of actions performed, and the number of error, success, and warning messages received.

Details

Contains an icon indicating status.

Action

Displays the steps being performed.

Status

Displays the status of the action step.

Message

Contains any messages returned by the action steps.

Tuning Log

Contains information regarding this tuning session. To print this log, right-click the log, and then click





Print.

See Also

[View and Work with the Output from the Database Engine Tuning Advisor
dta Utility](#)

Reduce the Production Server Tuning Load

1/17/2018 • 4 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

Database Engine Tuning Advisor relies on the query optimizer to analyze a workload and to make tuning recommendations. Performing this analysis on the production server adds to the server load and can hurt server performance during the tuning session. You can reduce the impact to the server load during a tuning session by using a test server in addition to the production server.

How Database Engine Tuning Advisor Uses a Test Server

The traditional way to use a test server is to copy all of the data from your production server to your test server, tune the test server, and then implement the recommendation on your production server. This process eliminates the performance impact on your production server, but nevertheless is not the optimal solution. For example, copying large amounts of data from the production to the test server can consume substantial amounts of time and resources. In addition, test server hardware is seldom as powerful as the hardware that is deployed for production servers. The tuning process relies on the query optimizer, and the recommendations it generates are based in part on the underlying hardware. If the test and production server hardware are not identical, the Database Engine Tuning Advisor recommendation quality is diminished.

To avoid these problems, Database Engine Tuning Advisor tunes a database on a production server by offloading most of the tuning load onto a test server. It does this by using the production server hardware configuration information and without actually copying the data from the production server to the test server. Database Engine Tuning Advisor does not copy actual data from the production server to the test server. It only copies the metadata and necessary statistics.

The following steps outline the process for tuning a production database on a test server:

1. Make sure that the user who wants to use the test server exists on both servers.

Before you start, make sure that the user who wants to use the test server to tune a database on the production server exists on both servers. This requires that you create the user and his or her login on the test server. If you are a member of the **sysadmin** fixed server role on both computers, this step is not necessary.

2. Tune the workload on the test server.

To tune a workload on a test server, you must use an XML input file with the **dta** command-line utility. In the XML input file, specify the name of your test server with the **TestServer** subelement in addition to specifying the values for the other subelements under the **TuningOptions** parent element.

During the tuning process, Database Engine Tuning Advisor creates a shell database on the test server. To create this shell database and tune it, Database Engine Tuning Advisor makes calls to the production server for the following:

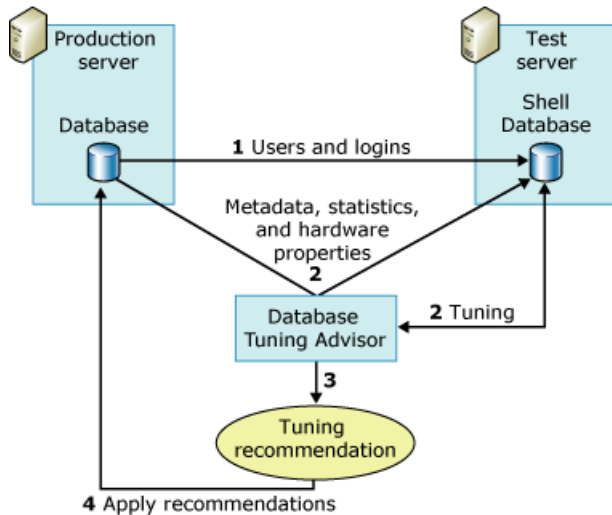
- a. Database Engine Tuning Advisor imports metadata from the production database to the test server shell database. This metadata includes empty tables, indexes, views, stored procedures, triggers, and so on. This makes it possible for the workload queries to execute against the test server shell database.
- b. Database Engine Tuning Advisor imports statistics from the production server so the query optimizer

can accurately optimize queries on the test server.

- c. Database Engine Tuning Advisor imports hardware parameters specifying the number of processors and available memory from the production server to provide the query optimizer with the information it needs to generate a query plan.

3. After Database Engine Tuning Advisor finishes tuning the test server shell database, it generates a tuning recommendation.
4. Apply the recommendation received from tuning the test server to the production server.

The following illustration shows the test server and production server scenario:



NOTE

The test server tuning feature is not supported in the Database Engine Tuning Advisor graphical user interface (GUI).

Example

First, make sure that the user who wants to perform the tuning exists on both the test and production servers.

After the user information is copied over to your test server, you can define your test server tuning session in the Database Engine Tuning Advisor XML input file. The following example XML input file illustrates how to specify a test server to tune a database with Database Engine Tuning Advisor.

In this example, the `MyDatabaseName` database is being tuned on `MyServerName`. The Transact-SQL script, `MyWorkloadScript.sql`, is used as the workload. This workload contains events that execute against `MyDatabaseName`. Most of the query optimizer calls to this database, which occur as part of the tuning process, are handled by the shell database that resides on `MyTestServerName`. The shell database is composed of metadata and statistics. This process results in the tuning overhead being offloaded to the test server. When Database Engine Tuning Advisor generates its tuning recommendation using this XML input file, it should consider indexes only (`<FeatureSet>IDX</FeatureSet>`), no partitioning, and need not keep any of the existing physical design structures in `MyDatabaseName`.


```
<?xml version="1.0" encoding="utf-16" ?>
<DTAXML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://schemas.microsoft.com/sqlserver/2004/07/dta">
  <DTAInput>
    <Server>
      <Name>MyServerName</Name>
      <Database>
        <Name>MyDatabaseName</Name>
      </Database>
    </Server>
    <Workload>
      <File>MyWorkloadScript.sql</File>
    </Workload>
    <TuningOptions>
      <TestServer>MyTestServerName</TestServer>
      <FeatureSet>IDX</FeatureSet>
      <Partitioning>NONE</Partitioning>
      <KeepExisting>NONE</KeepExisting>
    </TuningOptions>
  </DTAInput>
</DTAXML>
```





See Also

[Considerations for Using Test Servers](#)

[XML Input File Reference \(Database Engine Tuning Advisor\)](#)

Considerations for Using Test Servers

1/17/2018 • 5 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

Using a test server to tune a database on a production server is an important benefit of Database Engine Tuning Advisor. Using this feature, you can offload tuning overhead to a test server without copying the actual data over to the test server from the production server.

NOTE

The test server tuning feature is not supported in the Database Engine Tuning Advisor graphical user interface (GUI).

To use this feature successfully, review the considerations listed in the following sections.

Setting Up the Test Server/Production Server Environment

- The user who wants to use a test server to tune a database on a production server must exist on both servers, or this scenario will not work.
- The extended stored procedure, **xp_msver**, must be enabled to use the test server/production server scenario. Database Engine Tuning Advisor uses this extended stored procedure to fetch the number of processors and the available memory of the production server to use while tuning the test server. If **xp_msver** is not enabled, Database Engine Tuning Advisor assumes the hardware characteristics of the computer where Database Engine Tuning Advisor is running. If the hardware characteristics of the computer where Database Engine Tuning Advisor is running are not available, one processor and 1024 megabytes (MBs) of memory are assumed. This extended stored procedure is turned on by default when you install SQL Server. For more information, see [Surface Area Configuration](#) and [xp_msver \(Transact-SQL\)](#).
- Database Engine Tuning Advisor expects the editions of SQL Server to be the same on both the test server and the production server. If there are two different editions, the edition on the test server takes precedence. For example, if the test server is running SQL Server Standard, Database Engine Tuning Advisor will not include indexed views, partitioning, and online operations in its recommendations even if the production server is running SQL Server Enterprise.

About Test Server/Production Server Behavior

- Database Engine Tuning Advisor takes into account hardware differences between the production and the test server when creating recommendations. The recommendation is the same as though the tuning was done on the production server alone.
- Database Engine Tuning Advisor may impose some load on the production server for gathering metadata as well as creation of statistics necessary for tuning.
- Database Engine Tuning Advisor does not copy actual data from the production server to the test server. It only copies the metadata of the databases and the necessary statistics.
- All session information is stored in **msdb** on the production server. This allows you to exploit any available test server for tuning, and information about all sessions is available in one place (the production server).

Issues Related to the Shell Database

- After tuning, Database Engine Tuning Advisor should remove any metadata that it created on the test server during the tuning process. This includes the shell database. If you are performing a series of tuning sessions with the same production and test servers, you may want to retain this shell database to save time. In the XML input file, specify the **RetainShellIDB** subelement with the other sub elements under the **TuningOptions** parent element. Using these options causes Database Engine Tuning Advisor to retain the shell database. For more information, see [XML Input File Reference \(Database Engine Tuning Advisor\)](#).
- Shell databases may be left behind on the test server after a successful test server/production server tuning session even if you have not specified the **RetainShellIDB** subelement. These unwanted shell databases may interfere with subsequent tuning sessions and should be dropped before performing another test server/production server tuning session. In addition, if a tuning session exits unexpectedly, the shell databases on the test server and the objects within those databases may be left behind on the test server. You should also delete these databases and objects before starting a new test server/production server tuning session.

Issues Related to the Tuning Process





- The user must check the tuning log for tuning errors that result from differences between the production and test servers, and for errors that result from copying metadata from the production to the test server. For example, a user login may not exist on the test server. If a user login does not exist on the test server, those events in the workload that are issued by that user login may not be tunable. Use the Database Engine Tuning Advisor GUI to view the tuning log. For more information, see [View and Work with the Output from the Database Engine Tuning Advisor](#)
- If Database Engine Tuning Advisor cannot tune many events because objects are missing in the shell database that Database Engine Tuning Advisor creates on the test server, the user must check the tuning log. Events that cannot be tuned are listed in the log. To successfully tune the database on the test server, the user must create the missing objects in the shell database, and then start a new tuning session.
- If a database with the same name already exists on the test server, Database Engine Tuning Advisor does not copy metadata, but continues tuning and gathers statistics as necessary. This is useful if the user has already created a database on the test server and has copied the appropriate metadata before invoking Database Engine Tuning Advisor.
- If the DATE_CORRELATION_OPTIMIZATION option is turned on for a database on the production server, metadata and the data associated with this option are not completely scripted while tuning the test server. When tuning is performed for a test server/production server scenario, the following issues may apply:
 - Users can have different query plans on the servers for queries that use the DATE_CORRELATION_OPTIMIZATION option.
 - Database Engine Tuning Advisor may suggest dropping indexed views that enforce the DATE_CORRELATION_OPTIMIZATION option in the recommendation script.

Therefore, you may want to ignore any recommendations that Database Engine Tuning Advisor makes about the indexed views that hold correlation statistics because Database Engine Tuning Advisor knows their costs but not their benefits. Database Engine Tuning Advisor may not recommend selection of certain indexes such as clustered indexes on **datetime** columns, which could be beneficial when DATE_CORRELATION_OPTIMIZATION is enabled.

To determine if a view is based on correlation statistics, select the **is_date_correlation_view** column of the [sys.views](#) catalog view.

View and Work with the Output from the Database Engine Tuning Advisor

1/17/2018 • 23 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

When the Database Engine Tuning Advisor tunes databases, it creates summaries, recommendations, reports, and tuning logs. You can use the tuning log output to troubleshoot Database Engine Tuning Advisor tuning sessions. You can use the summaries, recommendations, and reports to determine whether you want to implement tuning recommendations or continue tuning until you achieve the query performance improvements that you need for your Microsoft SQL Server installation. For information about how to use the Database Tuning Advisor to create workloads and tune a database, see [Start and Use the Database Engine Tuning Advisor](#).

View Tuning Output

The following procedures describe how to view tuning recommendations, summaries, reports, and tuning logs by using the Database Engine Tuning Advisor GUI. For information about user interface options, see [User Interface Descriptions](#) later in this topic.

You can also use the GUI to view tuning output that is generated by the **dta** command-line utility.

NOTE

If you use the **dta** command-line utility and specify that output be written to an XML file by using the **-ox** argument, you can open and view the XML output file by clicking **Open File** on the **File** menu of SQL Server Management Studio. For more information, see [Use SQL Server Management Studio](#). For information about the **dta** command-line utility, see [dta Utility](#).

To view tuning recommendations with the Database Engine Tuning Advisor GUI

1. Tune a database using the Database Engine Tuning Advisor GUI or the **dta** command-line utility. For more information, see [Start and Use the Database Engine Tuning Advisor](#). If you want to use an existing tuning session, skip this step and proceed to Step 2.
2. Start the Database Engine Tuning Advisor GUI. For more information, see [Start and Use the Database Engine Tuning Advisor](#). If you want to view tuning recommendations for an existing tuning session, open it by double-clicking the session name in the **Session Monitor window**.

After the new tuning session has finished, or after the tool has loaded the existing session, the **Recommendations** page is displayed.

3. On the **Recommendations** page, click **Partition Recommendations** and **Index Recommendations** to view panes that display the tuning session results. If you did not specify partitioning when you set the tuning options for this session, the **Partition Recommendations** pane is empty.
4. In either the **Partition Recommendations** or the **Index Recommendations** pane, use the scroll bars to view all the information displayed in the grid.
5. Uncheck **Show existing objects** at the bottom of the **Recommendations** tabbed page. This causes the grid to display only those database objects that are referenced in the recommendation. Use the bottom scroll bar to view the right-most column in the recommendations grid, and click an item in the **Definition**

column to view or copy the Transact-SQL script that creates that object in your database.

6. If you want to save all of the Transact-SQL scripts that create or drop all database objects in this recommendation into one script file, click **Save Recommendations** on the **Actions** menu.

To view the tuning summary and reports with the Database Engine Tuning Advisor GUI

1. Tune a database using the Database Engine Tuning Advisor GUI or the **dta** command-line utility. For more information, see [Start and Use the Database Engine Tuning Advisor](#). If you want to use an existing tuning session, skip this step and proceed to step 2.
2. Start the Database Engine Tuning Advisor GUI. For more information, see [Start and Use the Database Engine Tuning Advisor](#). If you want to view tuning summaries and reports for an existing tuning session, open it by double-clicking the session name in the **Session Monitor**.
3. After the new tuning session has finished, or after the tool has loaded the existing session, click the **Reports** tab.
4. The **Tuning Summary** pane contains information about the tuning session. The information provided by the **Expected percentage improvement** and the **Space used by recommendation** items can be especially useful to decide whether you want to implement the recommendation.
5. In the **Tuning Reports** pane, click **Select report** to choose a tuning report to view.

To view tuning logs with the Database Engine Tuning Advisor GUI

1. Tune a database using the Database Engine Tuning Advisor GUI or the **dta** command-line utility. Make sure that you check **Save tuning log** on the **General** tab when you tune the workload. If you want to use an existing tuning session, skip this step and proceed to Step 2.
2. Start the Database Engine Tuning Advisor GUI. For more information, see [Start and Use the Database Engine Tuning Advisor](#). If you want to view tuning summaries and reports for an existing tuning session, open it by double-clicking the session name in the **Session Monitor** window.
3. After the new tuning session has finished, or after the tool has loaded the existing session, click the **Progress** tab. The **Tuning Log** pane displays the contents of the log. The log contains information about workload events that Database Engine Tuning Advisor was unable to analyze.

If all events in the tuning session were analyzed by Database Engine Tuning Advisor a message indicating that the tuning log is empty for the session displays. If **Save tuning log** was not checked on the **General** tab when the tuning session was originally run, a message displays indicating that.

Implement Tuning Recommendations

You can implement the Database Engine Tuning Advisor recommendations manually or automatically as part of the tuning session. If you want to examine the tuning results first before implementing them, use the Database Engine Tuning Advisor GUI. You can then use SQL Server Management Studio to manually run the Transact-SQL scripts that Database Engine Tuning Advisor generates as a result of analyzing a workload to implement the recommendations. If you do not need to examine the results before implementing them, you can use the **-a** option with the **dta** command prompt utility. This causes the utility to automatically implement the tuning recommendations after it analyzes your workload. The following procedures explain how to use both Database Engine Tuning Advisor interfaces to implement tuning recommendations.

To manually implement tuning recommendations with the Database Engine Tuning Advisor GUI

1. Tune a database using the Database Engine Tuning Advisor GUI or the **dta** command prompt utility. For more information, see [Start and Use the Database Engine Tuning Advisor](#). If you want to use an existing tuning session, skip this step and proceed to Step 2.
2. Start the Database Engine Tuning Advisor GUI. For more information, see [Start and Use the Database Engine Tuning Advisor](#). If you want to implement tuning recommendations for an existing tuning session,

open it by double-clicking the session name in **Session Monitor**.

3. After the new tuning session has finished, or after the tool has loaded the existing session, click **Apply Recommendations** on the **Actions** menu.
4. In the **Apply Recommendations** dialog box choose from **Apply now** or **Schedule for later**. If you choose **Schedule for later**, select the appropriate date and time.
5. Click **OK** to apply the recommendations.

To automatically implement tuning recommendations using the **dta** command prompt utility

1. Determine the database features (indexes, indexed views, partitioning) you want Database Engine Tuning Advisor to consider adding, removing, or retaining during analysis.

Keep the following considerations in mind before you begin tuning:

- When using a trace table as a workload, that table must exist on the same server where Database Engine Tuning Advisor is tuning. If you create the trace table on a different server, move it to the server where Database Engine Tuning Advisor is tuning.
- If a tuning session continues running longer than you had anticipated it would run, you can press CTRL+C to end the tuning session. Pressing CTRL+C under these circumstances forces **dta** to produce the best recommendation possible based on how much of the workload it has consumed, and does not waste the time that the tool has already used to tune the workload.

2. From a command prompt, enter the following:

```
dta -E -D DatabaseName -if WorkloadFile -s SessionName -a
```

where **-E** specifies that your tuning session uses a trusted connection (instead of a login ID and password), **-D** specifies the name of the database you want to tune or a comma-delimited list of multiple databases that the workload uses, **-if** specifies the name and path to a workload file, **-s** specifies a name for your tuning session, and **-a** specifies that you want the **dta** command prompt utility to automatically apply the tuning recommendations after the workload is analyzed without prompting you. For more information about using the **dta** command prompt utility to tune databases, see [Start and Use the Database Engine Tuning Advisor](#).

3. Press ENTER.

Perform Exploratory Analysis

The user-specified configuration feature of Database Engine Tuning Advisor enables database administrators to perform exploratory analysis. Using this feature, database administrators specify a desired physical database design to Database Engine Tuning Advisor, and then they can evaluate the performance effects of that design without implementing it. User-specified configuration is supported by both the Database Engine Tuning Advisor graphical user interface (GUI) and the command-line utility. However, the command-line utility provides the greatest flexibility.

If you use the Database Engine Tuning Advisor GUI, you can evaluate the effects of implementing a subset of a Database Engine Tuning Advisor tuning recommendation, but you cannot add hypothetical physical design structures for Database Engine Tuning Advisor to evaluate.

The following procedures explain how to use the user-specified configuration feature with both tool interfaces.

Using Database Engine Tuning Advisor GUI to Evaluate Tuning Recommendations

The following procedure describes how to evaluate a recommendation that is generated by Database Engine Tuning Advisor, but the GUI does not enable you to specify new physical design structures for evaluation.

To evaluate tuning recommendations with the Database Engine Tuning Advisor GUI

1. Use the Database Engine Tuning Advisor GUI to tune a database. For more information, see [Start and Use the Database Engine Tuning Advisor](#). If you want to evaluate an existing tuning session, double-click it in **Session Monitor**.
2. On the **Recommendations** tab, clear the recommended physical design structures that you do not want to use.
3. On the **Actions** menu, click **Evaluate Recommendations**. A new tuning session is created for you.
4. Type the new **Session name**. To view the physical database design structure configuration that you are evaluating, choose **Click here to see the configuration section**, in the **Description** area at the bottom of the Database Engine Tuning Advisor application window.
5. Click the **Start Analysis** button on the toolbar. When Database Engine Tuning Advisor is finished, you can view the results on the **Recommendations** tab.

Using Database Engine Tuning Advisor GUI to Export Tuning Session Results for "What-if" Tuning Analysis

The following procedure describes how to export Database Engine Tuning Advisor tuning session results to an XML file, which you can edit, and then tune it with the **dta** command-line utility. This enables you to perform tuning analysis on hypothetical new physical design structures without incurring the overhead of implementing them in your database before you find out whether they produce the performance improvements that you need. Using the Database Engine Tuning Advisor GUI to initially tune your database and then exporting the tuning results to an **.xml** file is a good way for users who are new to XML to use the flexibility of the Database Engine Tuning Advisor XML schema to perform "what-if" analysis.

To export tuning session results from the Database Engine Tuning Advisor GUI for "what-if" analysis with the dta command-line utility

1. Use the Database Engine Tuning Advisor GUI to tune a database. For more information, see [Start and Use the Database Engine Tuning Advisor](#). If you want to evaluate an existing tuning session, double-click it in the **Session Monitor**.
2. On the **File** menu, click **Export Session Results** and save it as an XML file.
3. Open the XML file created in Step 2 in your favorite XML editor, text editor, or in SQL Server Management Studio. Scroll down to the **Configuration** element. Copy and paste the **Configuration** element section into an XML input file template after the **TuningOptions** element. Save this XML input file.
4. In the new XML input file that you created in Step 3, specify any tuning options you want in the **TuningOptions** element, edit the **Configuration** element section (add or delete the physical design structures as appropriate for your analysis), save the file, and validate it against the Database Engine Tuning Advisor XML schema. For information about editing this XML file, see [XML Input File Reference \(Database Engine Tuning Advisor\)](#).
5. Use the XML file that you created in Step 4 as input to the **dta** command line utility. For information about using XML input files with this tool, see the section "Tune a Database by Using the dta Utility" in [Start and Use the Database Engine Tuning Advisor](#).

Using the User-specified Configuration Feature with the dta Command Line Utility

If you are an experienced XML developer, you can create a Database Engine Tuning Advisor XML input file in which you can specify a workload and a hypothetical configuration of physical database design structures, such as indexes, indexed views, or partitioning. Then you can use the **dta** command-line utility to analyze the effects this hypothetical configuration has on query performance for your database. The following procedure explains this process step by step:

To use the user-specified configuration feature with the dta command line utility

1. Create a tuning workload. For information about performing this task, see [Start and Use the Database Engine Tuning Advisor](#).

2. Copy and paste the [XML Input File Sample with User-specified Configuration \(DTA\)](#) into your XML editor or a text editor. Use this sample to create an XML input file for your tuning session. For information about performing this task, see the section "Create XML Input Files" in [Start and Use the Database Engine Tuning Advisor](#).
3. Edit the **TuningOptions** and the **Configuration** elements in the sample XML input file. In the **TuningOptions** element, specify what physical design structures you want Database Engine Tuning Advisor to consider during the tuning session. In the **Configuration** element, specify the physical design structures that match the hypothetical configuration of physical database design structures that you want Database Engine Tuning Advisor to analyze. For information about what attributes and child elements you can use with the **TuningOptions** and the **Configuration** parent elements, see [XML Input File Reference \(Database Engine Tuning Advisor\)](#).
4. Save the input file with an **.xml** extension.
5. Validate the XML input file you saved in Step 4 against the Database Engine Tuning Advisor XML schema. This schema is installed at the following location when you install Microsoft SQL Server:

```
C:\Program Files\Microsoft SQL Server\100\Tools\Binn\schemas\sqlserver\2004\07\dta\dta.schema.xsd
```

The Database Engine Tuning Advisor XML schema is also available online at <http://schemas.microsoft.com/sqlserver/2004/07/dta>.

6. After creating a workload and an XML input file, you are ready to submit the input file to the **dta** command-line utility for analysis. Make sure that you specify an XML output file name for the **-ox** utility argument. This creates an XML output file with a recommended configuration specified in the **Configuration** element. If you want to run Database Engine Tuning Advisor again to check another hypothetical configuration that is based on the output, you can copy and paste the **Configuration** element contents from the output file into a new or your original XML input file. For information about using an XML input file with the **dta** utility, see the section "Tune a Database by Using the dta Utility" in [Start and Use the Database Engine Tuning Advisor](#).

After tuning is finished, either use the Database Engine Tuning Advisor GUI to view the tuning reports, or open the XML output file to view the **TuningSummary** and the **Configuration** elements to view the Database Engine Tuning Advisor recommendations. For information about viewing the results of your tuning session, see [View Tuning Output](#) earlier in this topic. Also note that the XML output file may contain Database Engine Tuning Advisor analysis reports.

7. Repeat steps 6 and 7 until you create the hypothetical configuration that produces the query performance improvement that you need. Then you can implement the new configuration. For more information, see [Implement Tuning Recommendations](#) earlier in this topic.

Review, Evaluate, and Clone Tuning Sessions

The Database Engine Tuning Advisor creates a new tuning session each time that you start analyzing the effects of a workload on your database or databases. You can use the **Session Monitor** in the Database Engine Tuning Advisor GUI to view or reload all tuning sessions that have run on a given instance of Microsoft SQL Server. Having all the existing tuning sessions available to review makes it easy to: clone sessions based on existing ones, edit existing tuning recommendations and then use Database Engine Tuning Advisor to evaluate the edited session, or perform tuning at regular intervals to monitor the physical design of your databases. For example, you may decide to tune database on a monthly schedule.

Before you can review any tuning sessions for an instance of SQL Server, you must create tuning sessions on the server instance by tuning workloads with the Database Engine Tuning Advisor. For more information, see [Start and Use the Database Engine Tuning Advisor](#).

Review Existing Tuning Sessions

Use the following steps to browse the existing tuning sessions on a given instance of SQL Server.

To review existing tuning sessions

1. Start the Database Engine Tuning Advisor GUI. For more information, see [Start and Use the Database Engine Tuning Advisor](#).
2. All existing tuning sessions are displayed in the upper half of the **Session Monitor** window. The number of sessions displayed depends on how many times you have tuned databases on this SQL Server instance. Use the scroll bars to view all tuning sessions.
3. Click a tuning session name once and its details appear in the lower half of the **Session Monitor** window.
4. Double-click a tuning session name and its information is loaded into Database Engine Tuning Advisor. After the session information loads, you can choose any of the tabs to view information about this tuning session.

Evaluate Existing Tuning Sessions As Hypothetical Configurations

Use the following steps to evaluate an existing tuning session. Evaluating an existing tuning session involves viewing and editing its recommendations, and then re-tuning. For example, you decide that you want to only create indexes on **table1**, so you delete the creation of indexed views and partitioning from an existing tuning recommendation. Then Database Engine Tuning Advisor creates a new tuning session and tunes the workload against your databases using the edited recommendations as a hypothetical configuration. This means that Database Engine Tuning Advisor tunes the workload against the databases as if the edited recommendations have been implemented, enabling you to perform limited "what-if" analysis. It is limited what-if analysis because you can only choose a subset of an existing recommendation when you use the Database Engine Tuning Advisor GUI. To perform full what-if analysis, specifying a completely new hypothetical configuration that is not a subset of any previous tuning session, you must use the Database Engine Tuning Advisor XML input file with the **dta** command-line utility.

To evaluate an existing tuning session

1. After starting Database Engine Tuning Advisor, double-click a tuning session in the upper half of the **Session Monitor**, which loads the session information into Database Engine Tuning Advisor.
2. Click the **Progress** tab to check the tuning log, which contains error information about any events in the workload that Database Engine Tuning Advisor could not tune. This information can help you evaluate the effectiveness of the workload.
3. If you would like to review the tuning results for this session further, click the **Reports** tab. There you can view the tuning summary or choose a tuning report from the **Select report** list.
4. Click the **Recommendations** tab to view the tuning recommendations.
5. If there are any recommendations that you are unsure about implementing, uncheck them.
6. On the **Actions** menu, click **Evaluate Recommendations**. Database Engine Tuning Advisor creates a new tuning session that uses the edited recommendation as a hypothetical configuration. To view the hypothetical configuration in XML, choose **Click here to see the configuration section**.
7. On the **General** tab, type a **Session name**, and make sure the correct **Workload** is specified.
8. On the **Tuning Options** tab, you can specify a tuning time, or any of the **Advanced Options**.
9. Click the **Start Analysis** button on the toolbar. Database Engine Tuning Advisor starts tuning the databases using the hypothetical configuration. When Database Engine Tuning Advisor finishes, you can view the results of this session as you normally would for any session.

Clone Existing Tuning Sessions

You can create new tuning sessions based on existing sessions by choosing the cloning option in Database Engine

Tuning Advisor. When you use the cloning option, you base a new tuning session on an existing session. Then you can change the tuning options for the new session as needed. When you evaluate an existing session as described in the previous procedure, Database Engine Tuning Advisor also creates a new tuning session, but you cannot change the tuning options.

To create new tuning sessions by cloning existing sessions

1. After starting Database Engine Tuning Advisor, double-click a tuning session in the upper half of the **Session Monitor**, which loads the session information into Database Engine Tuning Advisor.
2. On the **Actions** menu, click **Clone Session**.
3. On the **General** tab, type a **Session name**, and make sure the correct **Workload** is specified.
4. On the **Tuning Options** tab, you can specify a tuning time, the physical design structures Database Engine Tuning Advisor should consider creating, and what it should consider dropping in its recommendation.
5. Click **Advanced Options** if you want to set a space limit for recommendations, a maximum number of columns per index, and whether you want Database Engine Tuning Advisor to generate recommendations that can be implemented while SQL Server is online.
6. Click the **Start Analysis** button on the toolbar to analyze the effects of the workload like any other tuning session. When Database Engine Tuning Advisor finishes, you can view the results of this session as you normally would for any session.

User Interface Descriptions

Sessions Monitor

Session Monitor displays information about sessions that are opened in the Database Engine Tuning Advisor. To display information about the session in the property window, select a session name in **Session Monitor**.

Recommendations Tab

The **Recommendations** tab appears after Database Engine Tuning Advisor completes analysis of a workload. This grid contains the recommendations for each object considered. Partition recommendations, if any, are presented in the upper grid, and the index recommendations are presented in the lower grid. The grids do not appear if there are no recommendations.

The **Definition** column contains the definition of the recommended partition or index as a hyperlink. This column is usually too narrow to see the entire definition. Click the hyperlink to display a dialog box containing the full definition and the **Copy to Clipboard** button.

Partition Recommendations

Database Name

The database containing the objects recommended to be modified.

Recommendation

The action recommended to improve performance. Possible values are Create and Drop.

Target of Recommendation

The partition function or scheme affected by the recommendation. The icon in this column reflects the recommendation to drop or add the **Target of Recommendation** and whether it is a partition function or scheme.

Details

A description of the **Target of Recommendation**. Possible values include a range for partition functions, or blank for partition schemes.

No. of Partitions

The number of partitions defined by the recommended partitioning functions. When this function is used with a

scheme and then applied to a table, the data in the table is divided into that many partitions.

Definition

The definition of the **Target of Recommendation**. Click the column to open the SQL Script Preview dialog box, with a script for the recommended action.

Index Recommendations

Database Name

The database containing the objects recommended to be modified.

Object Name

The table related to the recommendation.

Recommendation

The action recommended to improve performance. Possible values are Create and Drop.

Target of Recommendation

The index or view affected by the recommendation. The icon in this column reflects the recommendation to drop or add the **Target of Recommendation**.

Details

A description of the **Target of Recommendation**. Possible values include clustered, indexed view, or blank indicating a non-clustered index. Also indicates whether the index is unique.

Partition Scheme

The partition scheme is provided in this column if partitioning is recommended.

Size (KB)

The expected size of the new object that is being recommended. If this column is blank, click **See Reports for sizes of existing objects**.

Definition

The definition of the **Target of Recommendation**. Click the column to open the SQL Script Preview dialog box, with a script for the recommended action.

Show existing objects

Select to show all existing objects in the grid, even if no recommendations related to the objects are made by Database Engine Tuning Advisor.

See Reports for sizes of existing objects

Select to view reports that provide the size of existing objects in the recommendations grid.

Actions Menu/Apply Recommendations Options

After a workload has been analyzed and recommendations have been presented, on the **Actions** menu, click **Apply Recommendations** to open the **Apply Recommendations** dialog box.

Apply now

Generate a script for the recommendations, and runs the script to implement the recommendations.

Schedule for later

Generate a script for the recommendations, and save the actions as a SQL Server Agent job.

Date

Specify the date you want to run the SQL Server Agent job to apply the recommendations.

Time

Specify the time you want to run the SQL Server Agent job to apply the recommendations.

Reports Tab Options

The **Reports** tab appears after Database Engine Tuning Advisor completes analysis of a workload.

Tuning Summary

Displays a summary of the Database Engine Tuning Advisor recommendations.

Date

The date that Database Engine Tuning Advisor created the report.

Time

The time that Database Engine Tuning Advisor created the report.

Server

The server that was the target of the Database Engine Tuning Advisor workload.

Databases to tune

The database affected by Database Engine Tuning Advisor recommendations.

Workload file

Appears when the workload is a file.

Workload table

Appears when the workload is a SQL Server table.

Workload

Appears when the workload has been imported from the Query Editor in SQL Server Management Studio.

Maximum tuning time

The maximum time configured to be available for the Database Engine Tuning Advisor analysis.

Time taken for tuning

The time actually used by Database Engine Tuning Advisor to analyze the workload.

Expected percentage improvement

The percentage improvement expected with the target workload if all Database Engine Tuning Advisor recommendations are implemented.

Maximum space for recommendation (MB)

The maximum space considered for the recommendations. This value is configured before the analysis is made, using the **Advanced Options** button, on the **Tuning Options** tab.

Space used currently (MB)

The space currently used by indexes in the database analyzed.

Space used by recommendation (MB)

The approximate space expected to be used by indexes if all the Database Engine Tuning Advisor recommendations are implemented.

Number of events in workload

Number of events contained in the workload.

Number of events tuned

Number of events in the workload that were tuned. If an event cannot be tuned, it is listed in the tuning log, which is available on the **Progress** tab.

Number of statements tuned

Number of statements in the workload that were tuned. If a statement cannot be tuned, it is listed in the tuning log which is available on the **Progress** tab.

Percent SELECT statements in the tuned set

Percentage of tuned statements that are SELECT statements. Only appears if there are SELECT statements that

have been tuned.

Percent UPDATE statements in the tuned set

Percentage of tuned statements that are UPDATE statements. Only appears if there are UPDATE statements that have been tuned.

Number of indexes recommended to be [created | dropped]

Recommended number of indexes to be created or dropped on the tuned database. Only appears if indexes are part of the recommendation.

Number of indexes on views recommended to be [created | dropped]

Recommended number of indexes on views to be created or dropped on the tuned database. Only appears if indexes on views are part of the recommendation.

Number of statistics recommended to be created

Recommended number of statistics to be created on the tuned database. Only appears if statistics are recommended.

Select Report





See the details of the selected report. The columns in the grid vary with each report.

See Also

[Start and Use the Database Engine Tuning Advisor
dta Utility](#)

Columnstore index recommendations in Database Engine Tuning Advisor (DTA)

1/17/2018 • 1 min to read • [Edit Online](#)

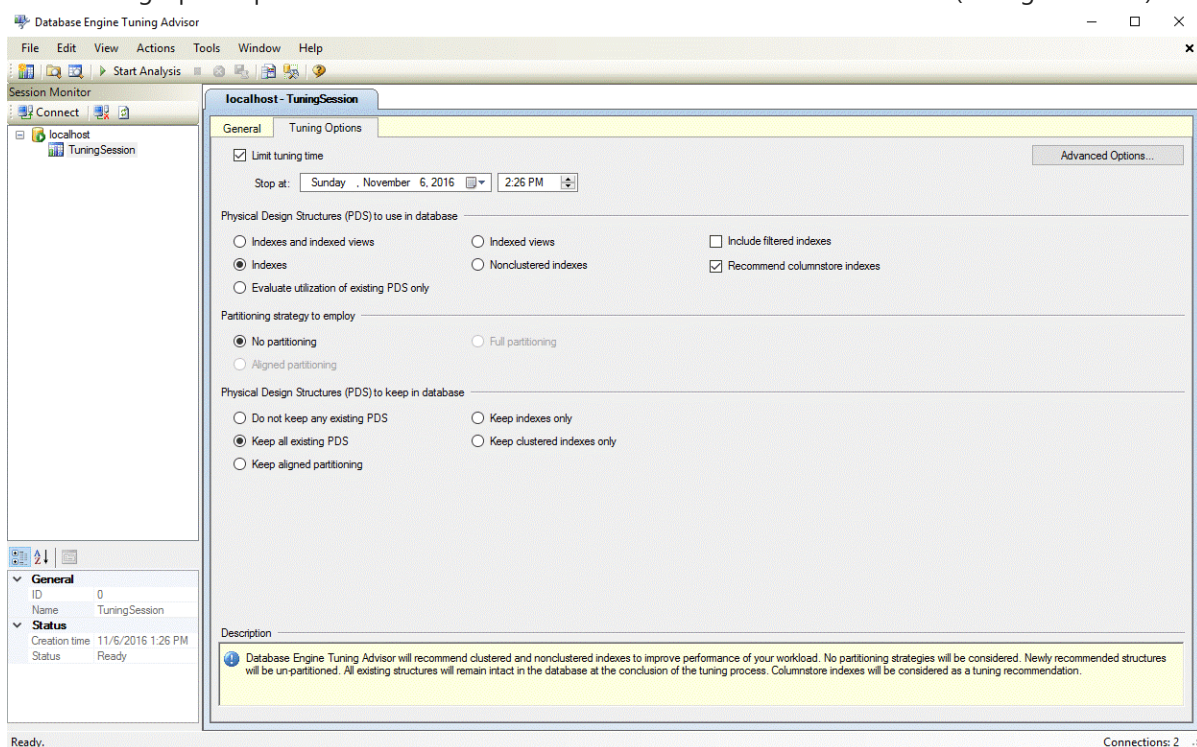
THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

Data warehousing and analytic workloads can greatly benefit from [columnstore indexes](#) as well as traditional rowstore indexes. The choice of which rowstore and columnstore indexes to build for your database is dependent on your application's workload. In SQL Server 2016, the [Database Engine Tuning Advisor \(DTA\)](#) can analyze your workload and recommend an appropriate combination of rowstore and columnstore indexes to build on the database.

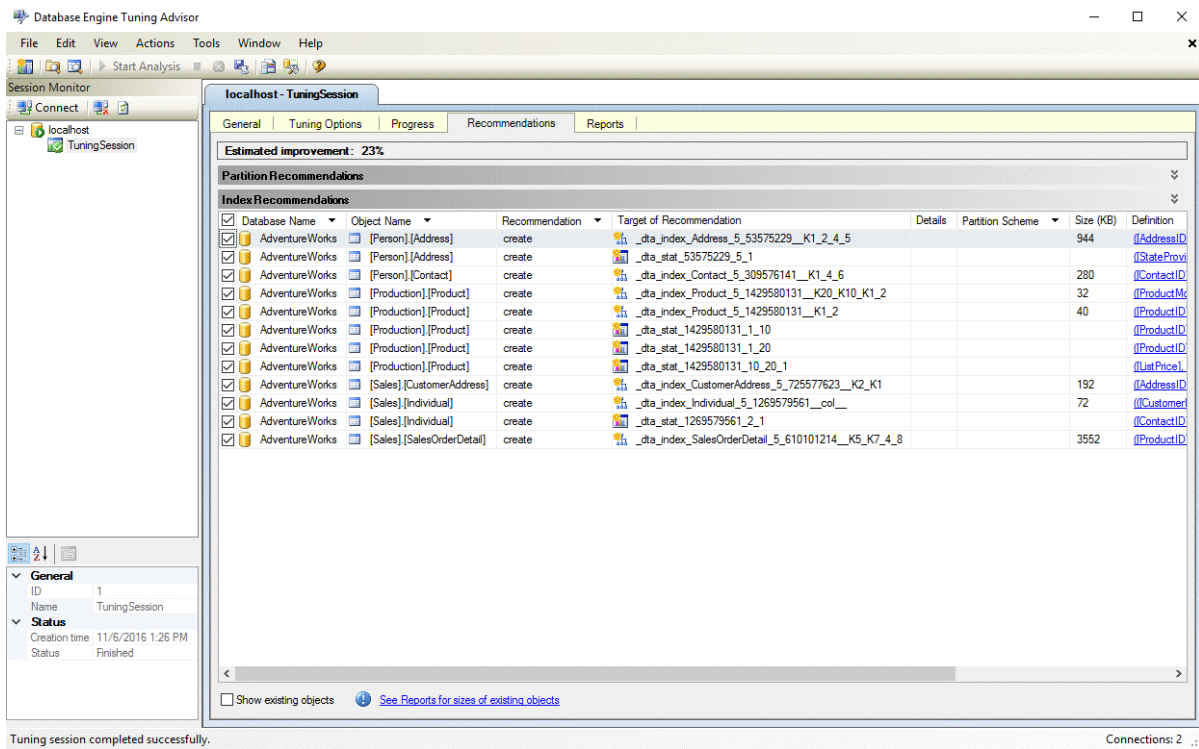
This feature is available with SQL Server Management Studio version **16.4** or higher.

How to enable columnstore index recommendations in Database Engine Tuning Advisor GUI

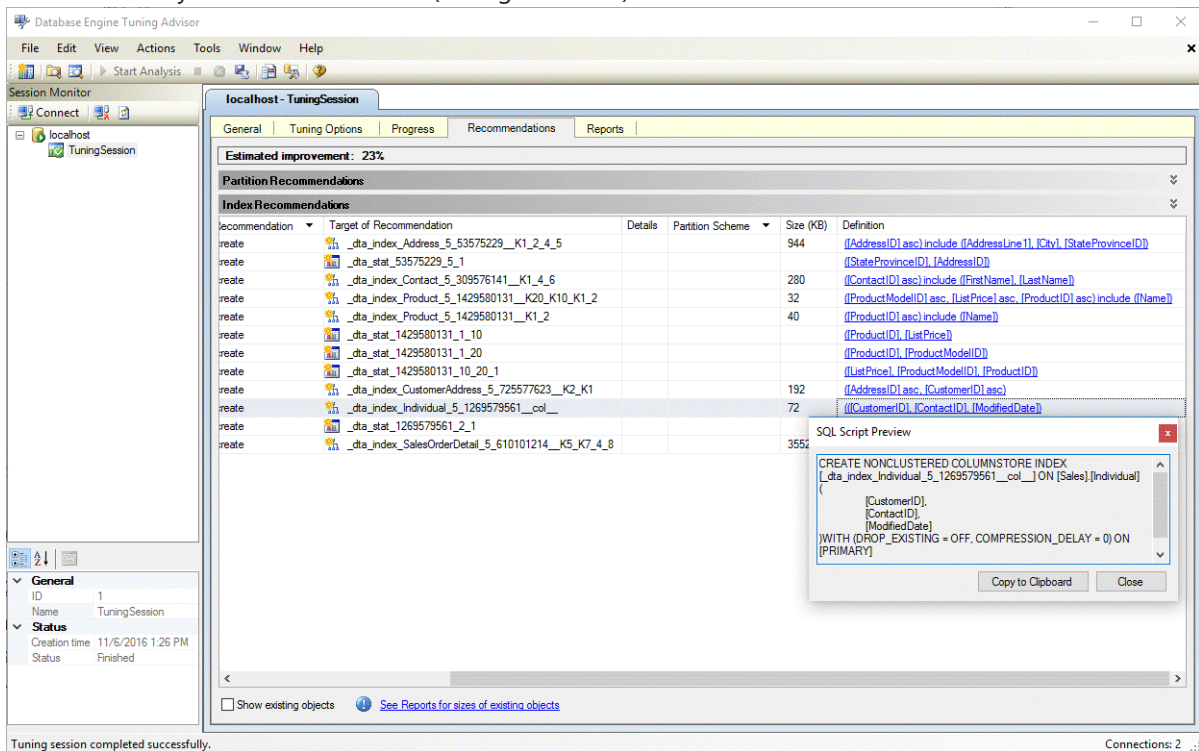
1. Launch Database Engine Tuning Advisor and open a new tuning session.
2. Select database(s) and workload for tuning in the **General** pane.
3. In the Tuning Options pane select the checkbox **Recommend columnstore indexes** (see figure below).



4. Select other tuning options and click on **Start Analysis** button.
5. Once tuning is complete, view all recommendations including any columnstore indexes in the **Recommendations** pane (see figure below).



6. Click on the **Definition** hyperlink to view the SQL Data Definition Language (DDL) statement that can create the recommended index. By default, DTA uses the suffix **col** in the name of columnstore indexes to make it easier to identify columnstore indexes (see figure below).



How to enable columnstore index recommendations in dta.exe Utility

To enable columnstore recommendations when using dta.exe command line utility, use the **-fc** command line parameter.





For more information on dta.exe command line utility, see [dta Utility](#)

See Also

[Columnstore Indexes Guide](#)

Tuning Database Using Workload from Query Store

1/17/2018 • 2 min to read • [Edit Online](#)

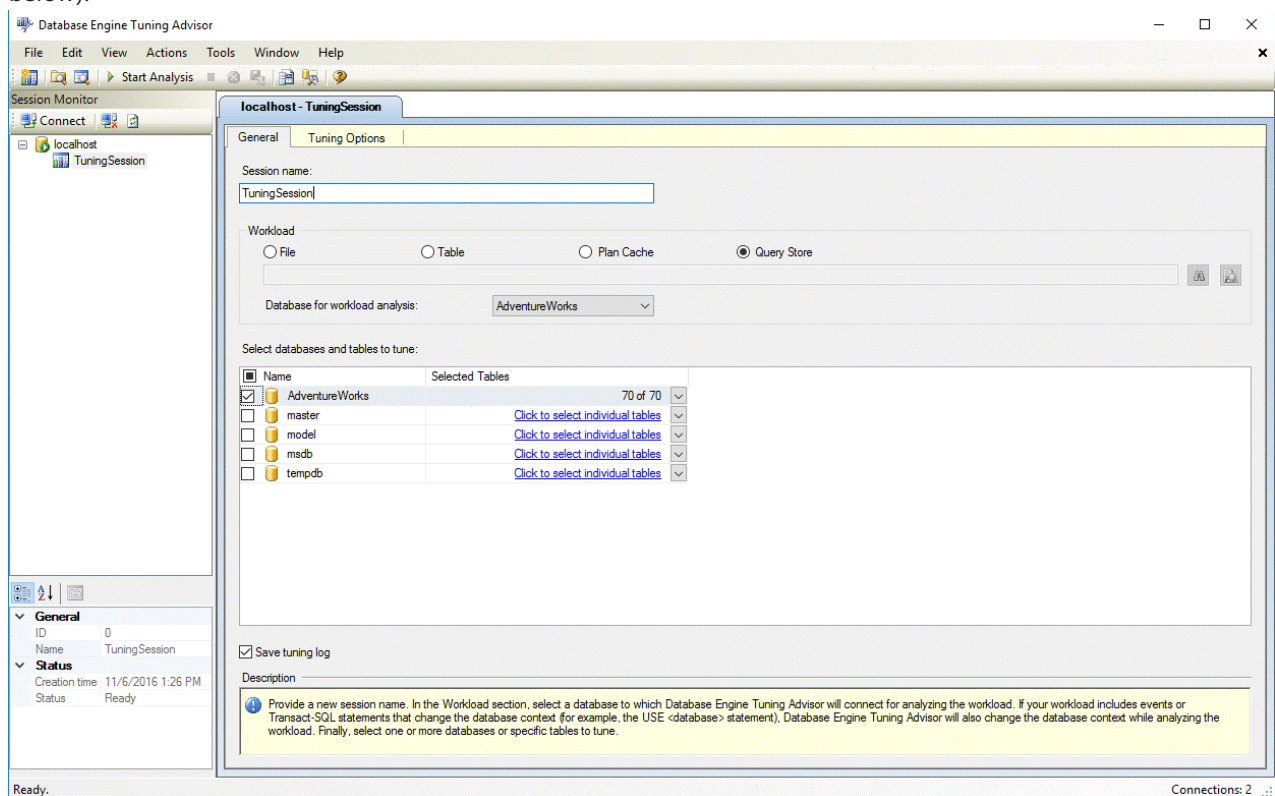
THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

The [Query Store](#) feature in SQL Server automatically captures a history of queries, plans, and runtime statistics, and persists this information in the database. The [Database Engine Tuning Advisor \(DTA\)](#) supports a new option to use the Query Store to automatically select an appropriate workload for tuning. For many users, this can take away the need to explicitly collect a workload for tuning. This feature is only available if the database has the Query Store feature turned on.

This feature is available with SQL Server Management Studio version **16.4** or higher.

How To Tune a Workload from Query Store in Database Engine Tuning Advisor GUI

From the DTA GUI, select the radio button **Query Store** in the **General** pane to enable this feature (see figure below).



How To Tune a Workload from Query Store in dta.exe command line Utility

From the command line (dta.exe), choose the **-iq** option to select the workload from Query Store.

There are two additional options available via the command line that helps tune the behavior of DTA when selecting the workload from Query Store. These options not available via the GUI:

1. Number of workload events to tune: This option, specified using **-n** command line argument, allows the user to control how many events from the Query Store are tuned. By default, DTA uses a value of 1000 for

this option. Note that DTA always chooses the most expensive events by total duration.

2. Time windows of events to tune: Since the Query Store may contain queries that have executed a long time ago, this option allows the user to specify a past time window (in hours) when a query must have executed for it to be considered by DTA for tuning. This option is specified using **-I** command line argument.

See [dta Utility](#) for more information.

Difference between using Workload from Query Store and Plan Cache

The difference between the Query Store and Plan Cache options is that the former contains a longer history of queries that have executed against the database, persisted across server restarts. On the other hand, the Plan Cache only contains a subset of recently-executed queries whose plans are cached in memory. When the server restarts, the entries in the Plan Cache are discarded.

See Also

[Database Engine Tuning Advisor](#)





[Tutorial: Database Engine Tuning Advisor](#)

[How Query Store Collects Data](#)

[Query Store Best Practices](#)

Performance Improvements using DTA recommendations

1/17/2018 • 3 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

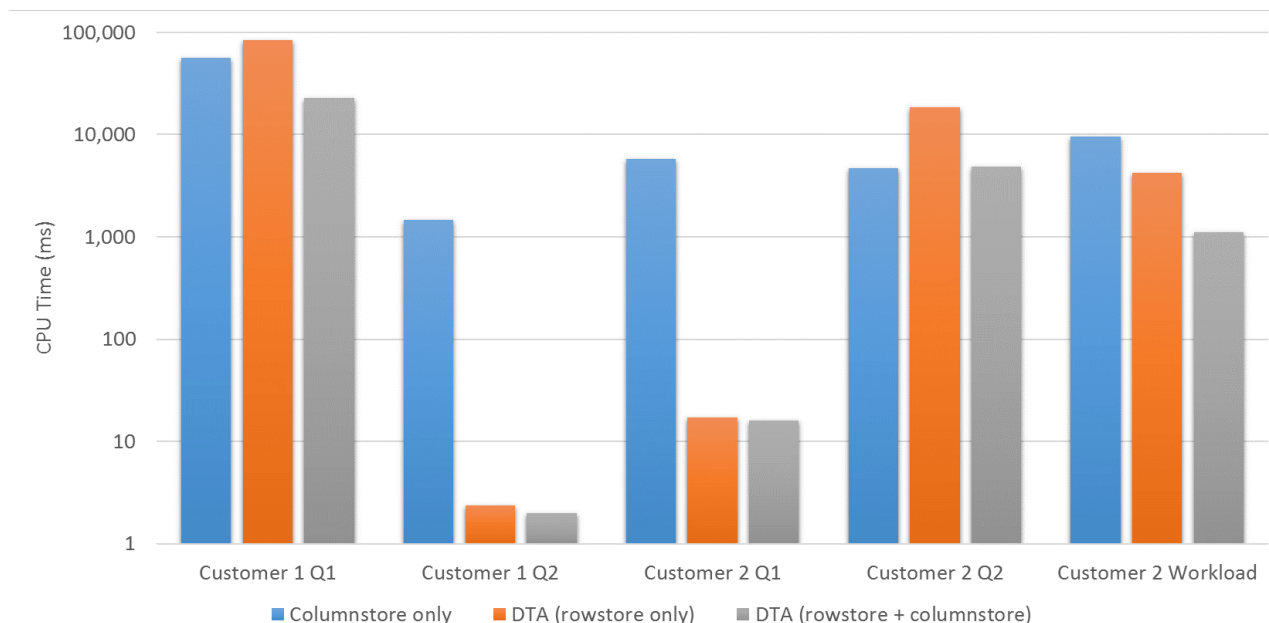
The performance of data warehousing and analytic workloads can greatly benefit from **columnstore** indexes, particularly for queries that need to scan large tables. **Rowstore** (B+-tree) indexes are most effective for queries that access relatively small amounts of data searching for a particular value or range of values. Since rowstore indexes can deliver rows in sorted order, they can also reduce the cost of sorting in query execution plans. Therefore, the choice of which combination of rowstore and columnstore indexes to build is dependent on your application's workload.

The Database Engine Tuning Advisor (DTA), starting in SQL Server 2016, can recommend a suitable **combination of rowstore and columnstore** indexes by analyzing a given database workload.

To demonstrate the benefits of DTA's recommendations on workload performance, we experimented with several real customer workloads. For each customer workload, we let DTA analyze individual queries as well as the full workload of queries. We consider three alternatives:

1. **Columnstore only:** Build only columnstore indexes for all tables without using DTA.
2. **DTA (rowstore only):** Run DTA with the option to recommend rowstore indexes only.
3. **DTA (rowstore + columnstore):** Run DTA with the option to recommend both rowstore and columnstore indexes.

In each case, we then implemented the recommended indexes. We report the CPU Time (in milliseconds) averaged over multiple runs of the query or the workload. The figure below plots the CPU time in milliseconds for workloads across two different customer databases. Note that the y-axis (CPU Time) uses a logarithmic scale.



Need for mixed physical designs: The first set of bars corresponding to Customer 1 Query 1. DTA (rowstore + columnstore) recommends a set of four columnstore and six rowstore indexes which results in 2.5X – 4X lower CPU time compared to columnstore index only and DTA (rowstore only). This demonstrates the benefits of mixed

physical designs consisting of rowstore and columnstore indexes *even for a single query*.

Effectiveness of rowstore index recommendations: The second and third set of bars (corresponding to Customer 1 Query 2 and Customer 2 Query 1) are cases where the queries have selective filter predicates that benefit from suitable rowstore indexes. For both these queries, DTA (rowstore only) and DTA (rowstore + columnstore) recommends rowstore indexes only. These examples also show that even when DTA is invoked with the option to recommend columnstore indexes, its cost-based approach ensures that it recommends a columnstore index only if the workload can actually benefit from it.

Effectiveness of columnstore index recommendations: The fourth set of bars corresponding to Customer 2 Query 2 represents a case where the query scans large tables which would benefit from columnstore indexes. DTA (rowstore only) generates a recommendation whose CPU Time is higher compared to when columnstore indexes are present. DTA (rowstore + columnstore) recommends suitable columnstore indexes, thus matching the query execution performance of the columnstore only option.

Effectiveness of recommendations for workload with multiple queries: The final set of bars corresponding to the full workload for Customer 2 exemplifies DTA's ability to analyze multiple queries in the workload to recommend a suitable set of rowstore and columnstore indexes which can improve the overall workload's execution cost. DTA (rowstore + columnstore) recommends four columnstore indexes and tens of rowstore indexes that result in over an order of magnitude improvement for the workload when compared to the option that builds only columnstore indexes; and about 4X–5X improvement when compared to DTA (rowstore only).

In summary, the above examples illustrate DTA's ability to suitably leverage both rowstore and columnstore indexes supported in the SQL Server Database Engine, and recommend an appropriate combination of indexes that can significantly reduce CPU Time for the workload.

See Also

[Database Engine Tuning Advisor](#)

[Columnstore index recommendations in Database Engine Tuning Advisor \(DTA\)](#)

[Columnstore Indexes Guide](#)





[Columnstore Indexes for Data Warehousing](#)

[CREATE COLUMNSTORE INDEX \(Transact-SQL\)](#)

[CREATE INDEX \(Transact-SQL\)](#)

Plan Guides

1/17/2018 • 6 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

Plan guides let you optimize the performance of queries when you cannot or do not want to directly change the text of the actual query in SQL Server 2017. Plan guides influence the optimization of queries by attaching query hints or a fixed query plan to them. Plan guides can be useful when a small subset of queries in a database application provided by a third-party vendor are not performing as expected. In the plan guide, you specify the Transact-SQL statement that you want optimized and either an `OPTION` clause that contains the query hints you want to use or a specific query plan you want to use to optimize the query. When the query executes, SQL Server matches the Transact-SQL statement to the plan guide and attaches the `OPTION` clause to the query at run time or uses the specified query plan.

The total number of plan guides you can create is limited only by available system resources. Nevertheless, plan guides should be limited to mission-critical queries that are targeted for improved or stabilized performance. Plan guides should not be used to influence most of the query load of a deployed application.

NOTE

Plan guides cannot be used in every edition of Microsoft SQL Server. For a list of features that are supported by the editions of SQL Server, see [Features Supported by the Editions of SQL Server 2016](#). Plan guides are visible in any edition. You can also attach a database that contains plan guides to any edition. Plan guides remain intact when you restore or attach a database to an upgraded version of SQL Server.

Types of Plan Guides

The following types of plan guides can be created.

OBJECT plan guide

An OBJECT plan guide matches queries that execute in the context of Transact-SQL stored procedures, scalar user-defined functions, multi-statement table-valued user-defined functions, and DML triggers.

Suppose the following stored procedure, which takes the `@Country_region` parameter, is in a database application that is deployed against the **AdventureWorks2012** database:

```
CREATE PROCEDURE Sales.GetSalesOrderByCountry (@Country_region nvarchar(60))
AS
BEGIN
    SELECT *
    FROM Sales.SalesOrderHeader AS h, Sales.Customer AS c,
         Sales.SalesTerritory AS t
    WHERE h.CustomerID = c.CustomerID
          AND c.TerritoryID = t.TerritoryID
          AND CountryRegionCode = @Country_region
END;
```

Assume that this stored procedure has been compiled and optimized for `@Country_region = N'AU'` (Australia). However, because there are relatively few sales orders that originate from Australia, performance decreases when the query executes using parameter values of countries with more sales orders. Because the most sales orders originate in the United States, a query plan that is generated for `@Country_region = N'US'` would likely perform

better for all possible values of the `@Country_region` parameter.

You could address this problem by modifying the stored procedure to add the `OPTIMIZE FOR` query hint to the query. However, because the stored procedure is in a deployed application, you cannot directly modify the application code. Instead, you can create the following plan guide in the **AdventureWorks2012** database.

```
sp_create_plan_guide
@name = N'Guide1',
@stmt = N'SELECT *FROM Sales.SalesOrderHeader AS h,
        Sales.Customer AS c,
        Sales.SalesTerritory AS t
        WHERE h.CustomerID = c.CustomerID
              AND c.TerritoryID = t.TerritoryID
              AND CountryRegionCode = @Country_region',
@type = N'OBJECT',
@module_or_batch = N'Sales.GetSalesOrderByCountry',
@params = NULL,
@hints = N'OPTION (OPTIMIZE FOR (@Country_region = N''US''))';
```

When the query specified in the `sp_create_plan_guide` statement executes, the query is modified before optimization to include the `OPTIMIZE FOR (@Country = N''US'')` clause.

SQL plan guide

An SQL plan guide matches queries that execute in the context of stand-alone Transact-SQL statements and batches that are not part of a database object. SQL-based plan guides can also be used to match queries that parameterize to a specified form. SQL plan guides apply to stand-alone Transact-SQL statements and batches. Frequently, these statements are submitted by an application by using the [sp_executesql](#) system stored procedure. For example, consider the following stand-alone batch:

```
SELECT TOP 1 * FROM Sales.SalesOrderHeader ORDER BY OrderDate DESC;
```

To prevent a parallel execution plan from being generated on this query, create the following plan guide and set the `MAXDOP` query hint to `1` in the `@hints` parameter.

```
sp_create_plan_guide
@name = N'Guide2',
@stmt = N'SELECT TOP 1 * FROM Sales.SalesOrderHeader ORDER BY OrderDate DESC',
@type = N'SQL',
@module_or_batch = NULL,
@params = NULL,
@hints = N'OPTION (MAXDOP 1)';
```

IMPORTANT

The values that are supplied for the `@module_or_batch` and `@params` arguments of the `sp_create_plan_guide` statement must match the corresponding text submitted in the actual query. For more information, see [sp_create_plan_guide \(Transact-SQL\)](#) and [Use SQL Server Profiler to Create and Test Plan Guides](#).

SQL plan guides can also be created on queries that parameterize to the same form when the `PARAMETERIZATION` database option is `SET` to `FORCED`, or when a `TEMPLATE` plan guide is created specifying that a parameterized class of queries.

TEMPLATE plan guide

A `TEMPLATE` plan guide matches stand-alone queries that parameterize to a specified form. These plan guides are used to override the current `PARAMETERIZATION` database `SET` option of a database for a class of queries.

You can create a TEMPLATE plan guide in either of the following situations:

- The PARAMETERIZATION database option is SET to FORCED, but there are queries you want compiled according to the rules of [Simple Parameterization](#).
- The PARAMETERIZATION database option is SET to SIMPLE (the default setting), but you want [Forced Parameterization](#) to be tried on a class of queries.

Plan Guide Matching Requirements

Plan guides are scoped to the database in which they are created. Therefore, only plan guides that are in the database that is current when a query executes can be matched to the query. For example, if

AdventureWorks2012 is the current database and the following query executes:

```
SELECT FirstName, LastName FROM Person.Person;
```

Only plan guides in the **AdventureWorks2012** database are eligible to be matched to this query. However, if **AdventureWorks2012** is the current database and the following statements are run:

```
USE DB1;  
SELECT FirstName, LastName FROM Person.Person;
```

Only plan guides in `DB1` are eligible to be matched to the query because the query is executing in the context of `DB1`.

For SQL- or TEMPLATE-based plan guides, SQL Server matches the values for the @module_or_batch and @params arguments to a query by comparing the two values character by character. This means you must provide the text exactly as SQL Server receives it in the actual batch.

When @type = 'SQL' and @module_or_batch is set to NULL, the value of @module_or_batch is set to the value of @stmt. This means that the value for *statement_text* must be provided in the identical format, character-for-character, as it is submitted to SQL Server. No internal conversion is performed to facilitate this match.

When both a regular (SQL or OBJECT) plan guide and a TEMPLATE plan guide can apply to a statement, only the regular plan guide will be used.

NOTE

The batch that contains the statement on which you want to create a plan guide cannot contain a USE *database* statement.

Plan Guide Effect on the Plan Cache

Creating a plan guide on a module removes the query plan for that module from the plan cache. Creating a plan guide of type OBJECT or SQL on a batch removes the query plan for a batch that has the same hash value. Creating a plan guide of type TEMPLATE removes all single-statement batches from the plan cache within that database.

Related Tasks

TASK	TOPIC
Describes how to create a plan guide.	Create a New Plan Guide

TASK	TOPIC
Describes how to create a plan guide for parameterized queries.	Create a Plan Guide for Parameterized Queries
Describes how to control query parameterization behavior by using plan guides.	Specify Query Parameterization Behavior by Using Plan Guides
Describes how to include a fixed query plan in a plan guide.	Apply a Fixed Query Plan to a Plan Guide
Describes how to specify query hints in a plan guide.	Attach Query Hints to a Plan Guide
Describes how to view plan guide properties.	View Plan Guide Properties
Describes how to use SQL Server Profiler to create and test plan guides.	Use SQL Server Profiler to Create and Test Plan Guides
Describes how to validate plan guides.	Validate Plan Guides After Upgrade

See Also

[sp_create_plan_guide](#) (Transact-SQL)

[sp_create_plan_guide_from_handle](#) (Transact-SQL)





[sp_control_plan_guide](#) (Transact-SQL)

[sys.plan_guides](#) (Transact-SQL)

[sys.fn_validate_plan_guide](#) (Transact-SQL)

Create a New Plan Guide

1/17/2018 • 4 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

Plan guides influence query optimization by attaching query hints or a fixed query plan to them. In the plan guide, you specify the statement that you want optimized, and either an `OPTION` clause that contains query hints you want to use, or a specific query plan you want to use to optimize the query. When the query executes, the query optimizer matches the Transact-SQL statement to the plan guide and either attaches the `OPTION` clause to the query at run time or uses the specified query plan.

A Plan Guide applies either a fixed query plan, and/or query hints, to a query.

Limitations and restrictions

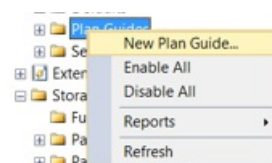
- The arguments to `sp_create_plan_guide` must be provided in the order that is shown. When you supply values for the parameters of **`sp_create_plan_guide`**, all parameter names must be specified explicitly, or none at all. For example, if **`@name`** = is specified, then **`@stmt`** = , **`@type`** = , and so on, must also be specified. Likewise, if **`@name`** = is omitted and only the parameter value is provided, the remaining parameter names must also be omitted, and only their values provided. Argument names are for descriptive purposes only, to help understand the syntax. SQL Server does not verify that the specified parameter name matches the name for the parameter in the position where the name is used.
- You can create more than one OBJECT or SQL plan guide for the same query and batch or module. However, only one plan guide can be enabled at any given time.
- Plan guides of type OBJECT cannot be created for an `@module_or_batch` value that references a stored procedure, function, or DML trigger that specifies the `WITH ENCRYPTION` clause or that is temporary.
- Trying to drop or modify a function, stored procedure, or DML trigger that is referenced by a plan guide, either enabled or disabled, causes an error. Trying to drop a table that has a trigger defined on it that is referenced by a plan guide also causes an error.

Permissions

To create a plan guide of type OBJECT, you need ALTER permission on the referenced object. To create a plan guide of type SQL or TEMPLATE, you need ALTER permission on the current database.

Create a plan guide using SSMS

1. Click the plus sign to expand the database in which you want to create a plan guide, and then click the plus sign to expand the **Programmability** folder.



2. Right-click the **Plan Guides** folder and select **New Plan Guide...**
3. In the **New Plan Guide** dialog box, in the **Name** box, enter the name of the plan guide.
4. In the **Statement** box, enter the Transact-SQL statement against which the plan guide is to be applied.

5. In the **Scope type** list, select the type of entity in which the Transact-SQL statement appears. This specifies the context for matching the Transact-SQL statement to the plan guide. Possible values are **OBJECT**, **SQL**, and **TEMPLATE**.
6. In the **Scope batch** box, enter the batch text in which the Transact-SQL statement appears. The batch text cannot include a USE ``database statement. The **Scope batch** box is only available when **SQL** is selected as a scope type. If nothing is entered in the scope batch box when SQL is the scope type, then the value of the batch text is set to the same value as is in the **Statement** box.
7. In the **Scope schema name** list, enter the name of the schema in which the object is contained. The **Scope schema name** box is only available when **Object** is selected as a scope type.
8. In the **Scope object name** box, enter the name of the Transact-SQL stored procedure, user-defined scalar function, multistatement table-valued function, or DML trigger in which the Transact-SQL statement appears. The **Scope object name** box is only available when **Object** is selected as a scope type.
9. In the **Parameters** box, enter the parameter name and data type of all parameters that are embedded in the Transact-SQL statement.

Parameters apply only when either of the following is true:

- The scope type is **SQL** or **TEMPLATE**. If **TEMPLATE**, parameters must not be NULL.
- The Transact-SQL statement is submitted by using sp_executesql and a value for the parameter is specified, or SQL Server internally submits a statement after parameterizing it.

10. In the **Hints** box, enter the query hints or query plan to be applied to the Transact-SQL statement. To specify one or more query hints, enter a valid OPTION clause.

11. Click **OK**.

The screenshot shows the 'New Plan Guide' dialog box. The 'Name' field is 'My new plan guide'. The 'Statement' field contains the following SQL query:

```
SELECT *
FROM [DBAnimals_DB].[dbo].[Ferret]
Where TypeOfFerret Like '%e%'
```

The 'Scope type' is set to 'Sql'. The 'Scope schema name' is 'dbo'. The 'Hints' field contains 'OPTION (MAXDOP 1)'. The 'OK' button is highlighted with a blue border.

Creat a plan guide using T-SQL

1. In **Object Explorer**, connect to an instance of Database Engine.





2. On the Standard bar, click **New Query**.
3. Copy and paste the following example into the query window and click **Execute**.

```
-- creates a plan guide named Guide1 based on a SQL statement
EXEC sp_create_plan_guide
    @name = N'Guide1',
    @stmt = N'SELECT TOP 1 *
              FROM Sales.SalesOrderHeader
              ORDER BY OrderDate DESC',
    @type = N'SQL',
    @module_or_batch = NULL,
    @params = NULL,
    @hints = N'OPTION (MAXDOP 1)';
```

For more information, see [sp_create_plan_guide \(Transact-SQL\)](#).

Create a Plan Guide for Parameterized Queries

1/17/2018 • 1 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

A TEMPLATE plan guide matches stand-alone queries that parameterize to a specified form.

The following example creates a plan guide that matches any query that parameterizes to a specified form, and directs SQL Server to force parameterization of the query. The following two queries are syntactically equivalent, but differ only in their constant literal values.

```
SELECT * FROM AdventureWorks2012.Sales.SalesOrderHeader AS h
INNER JOIN AdventureWorks2012.Sales.SalesOrderDetail AS d
    ON h.SalesOrderID = d.SalesOrderID
WHERE h.SalesOrderID = 45639;

SELECT * FROM AdventureWorks2012.Sales.SalesOrderHeader AS h
INNER JOIN AdventureWorks2012.Sales.SalesOrderDetail AS d
    ON h.SalesOrderID = d.SalesOrderID
WHERE h.SalesOrderID = 45640;
```

Here is the plan guide on the parameterized form of the query:

```
EXEC sp_create_plan_guide
    @name = N'TemplateGuide1',
    @stmt = N'SELECT * FROM AdventureWorks2012.Sales.SalesOrderHeader AS h
            INNER JOIN AdventureWorks2012.Sales.SalesOrderDetail AS d
                ON h.SalesOrderID = d.SalesOrderID
            WHERE h.SalesOrderID = @0',
    @type = N'TEMPLATE',
    @module_or_batch = NULL,
    @params = N'@0 int',
    @hints = N'OPTION(PARAMETERIZATION FORCED)';
```

In the previous example, the value for the `@stmt` parameter is the parameterized form of the query. The only reliable way to obtain this value for use in `sp_create_plan_guide` is to use the [sp_get_query_template](#) system stored procedure. The following script can be used both to obtain the parameterized query and then create a plan guide on it.

```
DECLARE @stmt nvarchar(max);
DECLARE @params nvarchar(max);
EXEC sp_get_query_template
    N'SELECT * FROM AdventureWorks2012.Sales.SalesOrderHeader AS h
    INNER JOIN AdventureWorks2012.Sales.SalesOrderDetail AS d
        ON h.SalesOrderID = d.SalesOrderID
    WHERE h.SalesOrderID = 45639;',
    @stmt OUTPUT,
    @params OUTPUT
EXEC sp_create_plan_guide N'TemplateGuide1',
    @stmt,
    N'TEMPLATE',
    NULL,
    @params,
    N'OPTION(PARAMETERIZATION FORCED)';
```





IMPORTANT

The value of the constant literals in the `@stmt` parameter passed to `sp_get_query_template` might affect the data type that is chosen for the parameter that replaces the literal. This will affect plan guide matching. You may have to create more than one plan guide to handle different parameter value ranges.

You can also use TEMPLATE plan guides together with SQL plan guides. For example, you can create a TEMPLATE plan guide to make sure that a class of queries is parameterized. You can then create an SQL plan guide on the parameterized form of that query.

Specify Query Parameterization Behavior by Using Plan Guides

1/17/2018 • 3 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

When the **PARAMETERIZATION** database option is set to **SIMPLE**, the SQL Server query optimizer may choose to parameterize the queries. This means that any literal values that are contained in a query are substituted with parameters. This process is referred to as simple parameterization. When **SIMPLE** parameterization is in effect, you cannot control which queries are parameterized and which queries are not. However, you can specify that all queries in a database be parameterized by setting the **PARAMETERIZATION** database option to **FORCED**. This process is referred to as forced parameterization.

You can override the parameterization behavior of a database by using plan guides in the following ways:

- When the **PARAMETERIZATION** database option is set to **SIMPLE**, you can specify that forced parameterization is attempted on a certain class of queries. You do this by creating a **TEMPLATE** plan guide on the parameterized form of the query, and specifying the **PARAMETERIZATION FORCED** query hint in the [sp_create_plan_guide](#) stored procedure. You can consider this kind of plan guide as a way to enable forced parameterization only on a certain class of queries, instead of all queries.
- When the **PARAMETERIZATION** database option is set to **FORCED**, you can specify that for a certain class of queries, only simple parameterization is attempted, not forced parameterization. You do this by creating a **TEMPLATE** plan guide on the force-parameterized form of the query, and specifying the **PARAMETERIZATION SIMPLE** query hint in **sp_create_plan_guide**.

Consider the following query on the **AdventureWorks2012** database:

```
SELECT pi.ProductID, SUM(pi.Quantity) AS Total
FROM Production.ProductModel AS pm
    INNER JOIN Production.ProductInventory AS pi
        ON pm.ProductModelID = pi.ProductID
WHERE pi.ProductID = 101
GROUP BY pi.ProductID, pi.Quantity HAVING SUM(pi.Quantity) > 50;
```

As a database administrator, you have determined that you do not want to enable forced parameterization on all queries in the database. However, you do want to avoid compilation costs on all queries that are syntactically equivalent to the previous query, but differ only in their constant literal values. In other words, you want the query to be parameterized so that a query plan for this kind of query is reused. In this case, complete the following steps:

1. Retrieve the parameterized form of the query. The only safe way to obtain this value for use in **sp_create_plan_guide** is by using the [sp_get_query_template](#) system stored procedure.
2. Create the plan guide on the parameterized form of the query, specifying the **PARAMETERIZATION FORCED** query hint.

IMPORTANT

As part of parameterizing a query, SQL Server assigns a data type to the parameters that replace the literal values, depending on the value and size of the literal. The same process occurs to the value of the constant literals passed to the **@stmt** output parameter of **sp_get_query_template**. Because the data type specified in the **@params** argument of **sp_create_plan_guide** must match that of the query as it is parameterized by SQL Server, you may have to create more than one plan guide to cover the complete range of possible parameter values for the query.

The following script can be used both to obtain the parameterized query and then create a plan guide on it:

```
DECLARE @stmt nvarchar(max);
DECLARE @params nvarchar(max);
EXEC sp_get_query_template
    N'SELECT pi.ProductID, SUM(pi.Quantity) AS Total
    FROM Production.ProductModel AS pm
    INNER JOIN Production.ProductInventory AS pi ON pm.ProductModelID = pi.ProductID
    WHERE pi.ProductID = 101
    GROUP BY pi.ProductID, pi.Quantity
    HAVING sum(pi.Quantity) > 50',
    @stmt OUTPUT,
    @params OUTPUT;
EXEC sp_create_plan_guide
    N'TemplateGuide1',
    @stmt,
    N'TEMPLATE',
    NULL,
    @params,
    N'OPTION(PARAMETERIZATION FORCED)';
```





Similarly, in a database in which forced parameterization is already enabled, you can make sure that the sample query, and others that are syntactically equivalent, except for their constant literal values, are parameterized according to the rules of simple parameterization. To do this, specify **PARAMETERIZATION SIMPLE** instead of **PARAMETERIZATION FORCED** in the **OPTION** clause.

NOTE

TEMPLATE plan guides match statements to queries submitted in batches that consist of a single statement only. Statements inside multistatement batches are not eligible to be matched by TEMPLATE plan guides.

Apply a Fixed Query Plan to a Plan Guide

1/17/2018 • 1 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

You can apply a fixed query plan to a plan guide of type OBJECT or SQL. Plan guides that apply a fixed query plan are useful when you know about an existing execution plan that performs better than the one selected by the optimizer for a particular query.





The following example creates a plan guide for a simple ad hoc SQL statement. The desired query plan for this statement is provided in the plan guide by specifying the XML Showplan for the query directly in the `@hints` parameter. The example first executes the SQL statement to generate a plan in the plan cache. For the purposes of this example, it is assumed that the generated plan is the desired plan and no additional query tuning is required. The XML Showplan for the query is obtained by querying the `sys.dm_exec_query_stats`, `sys.dm_exec_sql_text`, and `sys.dm_exec_text_query_plan` dynamic management views and is assigned to the `@xml_showplan` variable. The `@xml_showplan` variable is then passed to the `sp_create_plan_guide` statement in the `@hints` parameter. Or, you can create a plan guide from a query plan in the plan cache by using the [sp_create_plan_guide_from_handle](#) stored procedure.

```
USE AdventureWorks2012;
GO
SELECT City, StateProvinceID, PostalCode FROM Person.Address ORDER BY PostalCode DESC;
GO
DECLARE @xml_showplan nvarchar(max);
SET @xml_showplan = (SELECT query_plan
    FROM sys.dm_exec_query_stats AS qs
    CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) AS st
    CROSS APPLY sys.dm_exec_text_query_plan(qs.plan_handle, DEFAULT, DEFAULT) AS qp
    WHERE st.text LIKE N'SELECT City, StateProvinceID, PostalCode FROM Person.Address ORDER BY PostalCode
DESC;%');

EXEC sp_create_plan_guide
    @name = N'Guide1_from_XML_showplan',
    @stmt = N'SELECT City, StateProvinceID, PostalCode FROM Person.Address ORDER BY PostalCode DESC;',
    @type = N'SQL',
    @module_or_batch = NULL,
    @params = NULL,
    @hints = @xml_showplan;
GO
```


Attach Query Hints to a Plan Guide

1/17/2018 • 2 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

Any combination of valid query hints can be used in a plan guide. When a plan guide matches a query, the `OPTION` clause specified in the hints clause of a plan guide is added to the query before it compiles and optimizes. If a query that is matched to a plan guide already has an `OPTION` clause, the query hints specified in the plan guide replace those in the query. However, for a plan guide to match a query that already has an `OPTION` clause, you must include the `OPTION` clause of the query when you specify the text of the query to match in the `sp_create_plan_guide` statement. If you want the hints specified in the plan guide to be added to the hints that already exist on the query, instead of replacing them, you must specify both the original hints and the additional hints in the `OPTION` clause of the plan guide.

Caution

Plan guides that misuse query hints can cause compilation, execution, or performance problems. Plan guides should be used only by experienced developers and database administrators.





Common Query Hints Used in Plan Guides

Queries that can benefit from plan guides are generally parameter-based, and may be performing poorly because they use cached query plans whose parameter values do not represent a worst-case or most representative scenario. The `OPTIMIZE FOR` and `RECOMPILE` query hints can be used to address this problem. `OPTIMIZE FOR` instructs SQL Server to use a particular value for a parameter when the query is optimized. `RECOMPILE` instructs the server to discard a query plan after execution, forcing the query optimizer to recompile a new query plan the next time that the same query is executed. For an example, see [Plan Guides](#).

In addition, you can specify the table hints `INDEX`, `FORCESCAN`, and `FORCESEEK` as query hints. When specified as query hints, these hints behave like an inline table or view hint. The `INDEX` hint forces the query optimizer to use only the specified indexes to access the data in the referenced table or view. The `FORCESEEK` hint forces the optimizer to use only an index seek operation to access the data in the referenced table or view. These hints provide additional plan guide functionality and allow you to have more influence over the optimization of queries that use the plan guide.

View Plan Guide Properties

1/17/2018 • 2 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

You can view the properties of plan guides in SQL Server 2017 by using SQL Server Management Studio or Transact-SQL

In This Topic

- **Before you begin:**

[Security](#)

- **To view the properties of plan guides, using:**

[SQL Server Management Studio](#)

[Transact-SQL](#)

Before You Begin

Security

Permissions

The visibility of the metadata in catalog views is limited to securables that either a user owns or on which the user has been granted some permission.

Using SQL Server Management Studio

To view the properties of a plan guide

1. Click the plus sign to expand the database in which you want to view the properties of a plan guide, and then click the plus sign to expand the **Programmability** folder.
2. Click the plus sign to expand the **Plan Guides** folder.
3. Right-click the plan guide of which you want to view the properties and select **Properties**.

The following properties show in the **Plan Guide Properties** dialog box.

Hints

Displays the query hints or query plan to be applied to the Transact-SQL statement. When a query plan is specified as a hint, the XML Showplan output for the plan is displayed.

Is disabled

Displays the status of the plan guide. Possible values are **True** and **False**.

Name

Displays the name of the plan guide.

Parameters

When the scope type is SQL or TEMPLATE, displays the name and data type of all parameters that are embedded in the Transact-SQL statement.

Scope batch

Displays the batch text in which the Transact-SQL statement appears.

Scope object name

When the scope type is OBJECT, displays the name of the Transact-SQL stored procedure, user-defined scalar function, multistatement table-valued function, or DML trigger in which the Transact-SQL statement appears.

Scope schema name

When the scope type is OBJECT, displays the name of the schema in which the object is contained.

Scope type

Displays the type of entity in which the Transact-SQL statement appears. This specifies the context for matching the Transact-SQL statement to the plan guide. Possible values are **OBJECT**, **SQL**, and **TEMPLATE**.

Statement

Displays the Transact-SQL statement against which the plan guide is applied.

4. Click **OK**.

Using Transact-SQL

To view the properties of a plan guide





1. In **Object Explorer**, connect to an instance of Database Engine.
2. On the Standard bar, click **New Query**.
3. Copy and paste the following example into the query window and click **Execute**.

```
-- If a plan guide named "Guide1" already exists in the AdventureWorks2012 database, delete it.
USE AdventureWorks2012;
GO
IF OBJECT_ID(N'Guide1') IS NOT NULL
    EXEC sp_control_plan_guide N'DROP', N'Guide1';
GO
-- creates a plan guide named Guide1 based on a SQL statement
EXEC sp_create_plan_guide
    @name = N'Guide1',
    @stmt = N'SELECT TOP 1 *
              FROM Sales.SalesOrderHeader
              ORDER BY OrderDate DESC',
    @type = N'SQL',
    @module_or_batch = NULL,
    @params = NULL,
    @hints = N'OPTION (MAXDOP 1)';
GO
-- Gets the name, created date, and all other relevant property information on the plan guide created
above.
SELECT name AS plan_guide_name,
       create_date,
       query_text,
       scope_type_desc,
       OBJECT_NAME(scope_object_id) AS scope_object_name,
       scope_batch,
       parameters,
       hints,
       is_disabled
FROM sys.plan_guides
WHERE name = N'Guide1';
GO
```

For more information, see [sys.plan_guides \(Transact-SQL\)](#).

Use SQL Server Profiler to Create and Test Plan Guides

1/17/2018 • 3 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

When you are creating a plan guide, you can use SQL Server Profiler to capture the exact query text for use in the *statement_text* argument of the **sp_create_plan_guide** stored procedure. This helps make sure that the plan guide will be matched to the query at compile time. After the plan guide is created, SQL Server Profiler can also be used to test that the plan guide is, in fact, being matched to the query. Generally, you should test plan guides by using SQL Server Profiler to verify that your query is being matched to your plan guide.

Capturing Query Text by Using SQL Server Profiler

If you run a query and capture the text exactly as it was submitted to SQL Server by using SQL Server Profiler, you can create a plan guide of type SQL or TEMPLATE that will match the query text exactly. This makes sure that the plan guide is used by the query optimizer.

Consider the following query that is submitted by an application as a stand-alone batch:

```
SELECT COUNT(*) AS c
FROM Sales.SalesOrderHeader AS h
INNER JOIN Sales.SalesOrderDetail AS d
    ON h.SalesOrderID = d.SalesOrderID
WHERE h.OrderDate BETWEEN '20000101' and '20050101';
```

Suppose you want this query to execute using a merge join operation, but SHOWPLAN indicates that the query is not using a merge join. You cannot change the query directly in the application, so instead you create a plan guide to specify that the MERGE JOIN query hint be appended to the query at compile time.

To capture the text of the query exactly as SQL Server receives it, follow these steps:

1. Start a SQL Server Profiler trace, making sure that the **SQL:BatchStarting** event type is selected.
2. Have the application run the query.
3. Pause the SQL Server Profiler Trace.
4. Click the **SQL:BatchStarting** event that corresponds to the query.
5. Right-click and select **Extract Event Data**.

IMPORTANT

Do not try to copy the batch text by selecting it from the lower pane of the Profiler trace window. This might cause the plan guide that you create to not match the original batch.

6. Save the event data to a file. This is the batch text.
7. Open the batch text file in Notepad and copy the text to the copy and paste buffer.
8. Create the plan guide and paste the copied text inside the quotation marks (") specified for the **@stmt**

argument. You must escape any single quotation marks in the **@stmt** argument by preceding them with another single quotation mark. Be careful not to add or remove any other characters when you insert these single quotation marks. For example, the date literal '20000101' must be delimited as ''20000101''.

Here is the plan guide:

```
EXEC sp_create_plan_guide
    @name = N'MyGuide1',
    @stmt = N'<paste the text copied from the batch text file here>',
    @type = N'SQL',
    @module_or_batch = NULL,
    @params = NULL,
    @hints = N'OPTION (MERGE JOIN)';
```

Testing Plan Guides by Using SQL Server Profiler

To verify that a plan guide is being matched to a query, follow these steps:

1. Start a SQL Server Profiler trace, making certain that the **Showplan XML** event type is selected (located under the **Performance** node).
2. Have the application run the query.
3. Pause the SQL Server Profiler Trace.
4. Find the **Showplan XML** event for the affected query.

NOTE

The **Showplan XML for Query Compile** event cannot be used. **PlanGuideDB** does not exist in that event.





5. If the plan guide is of type OBJECT or SQL, verify that the **Showplan XML** event contains the **PlanGuideDB** and **PlanGuideName** attributes for the plan guide that you expected to match the query. Or, in the case of a TEMPLATE plan guide, verify that the **Showplan XML** event contains the **TemplatePlanGuideDB** and **TemplatePlanGuideName** attributes for the expected plan guide. This verifies that the plan guide is working. These attributes are contained under the **<StmtSimple>** element of the plan.

See Also

[sp_create_plan_guide \(Transact-SQL\)](#)

Validate Plan Guides After Upgrade

1/17/2018 • 1 min to read • [Edit Online](#)





THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

We recommend re-evaluating and testing plan guide definitions when you upgrade your application to a new release of SQL Server. Performance tuning requirements and plan guide matching behavior may change. Although an invalid plan guide will not cause a query to fail, the plan is compiled without using the plan guide and may not be the best choice. After upgrading a database to SQL Server 2017, we recommend that you perform the following tasks:

- Validate existing plan guides by using the [sys.fn_validate_plan_guide](#) function.
- Use extended events to monitor for misguided plans for some period of time by using the [Plan Guide Unsuccessful](#) event.

Delete a Plan Guide

1/17/2018 • 1 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

You can delete (drop) a plan guide in SQL Server 2017 by using SQL Server Management Studio or Transact-SQL. Using Transact-SQL, you can also delete all of the plan guides in a database.

In This Topic

- **Before you begin:**

[Security](#)

- **To delete a plan guide, using:**

[SQL Server Management Studio](#)

[Transact-SQL](#)

Before You Begin

Security

Permissions

Deleting an OBJECT plan guide requires ALTER permission on the object (for example: function, stored procedure) that is referenced by the plan guide. All other plan guides require ALTER DATABASE permission.

Using SQL Server Management Studio

To delete a plan guide

1. Click the plus sign to expand the database in which you want to delete a plan guide, and then click the plus sign to expand the **Programmability** folder.
2. Click the plus sign to expand the **Plan Guides** folder.
3. Right-click the plan guide you want to delete and select **Delete**.
4. In the **Delete Object** dialog box, ensure that the correct plan guide is selected and then click **OK**.

Using Transact-SQL

To delete a single plan guide

1. In **Object Explorer**, connect to an instance of Database Engine.
2. On the Standard bar, click **New Query**.
3. Copy and paste the following example into the query window and click **Execute**.

```

--Create a procedure on which to define the plan guide.
IF OBJECT_ID(N'Sales.GetSalesOrderByCountry', N'P') IS NOT NULL
    DROP PROCEDURE Sales.GetSalesOrderByCountry;
GO
CREATE PROCEDURE Sales.GetSalesOrderByCountry
    (@Country nvarchar(60))
AS
BEGIN
    SELECT *
    FROM Sales.SalesOrderHeader AS h
    INNER JOIN Sales.Customer AS c ON h.CustomerID = c.CustomerID
    INNER JOIN Sales.SalesTerritory AS t ON c.TerritoryID = t.TerritoryID
    WHERE t.CountryRegionCode = @Country;
END
GO
--Create the plan guide.
EXEC sp_create_plan_guide N'Guide3',
    N'SELECT *
    FROM Sales.SalesOrderHeader AS h
    INNER JOIN Sales.Customer AS c ON h.CustomerID = c.CustomerID
    INNER JOIN Sales.SalesTerritory AS t ON c.TerritoryID = t.TerritoryID
    WHERE t.CountryRegionCode = @Country',
    N'OBJECT',
    N'Sales.GetSalesOrderByCountry',
    NULL,
    N'OPTION (OPTIMIZE FOR (@Country = N''US''))';
GO
--Drop the plan guide.
EXEC sp_control_plan_guide N'DROP', N'Guide3';
GO

```

To delete all plan guides in a database

1. In **Object Explorer**, connect to an instance of Database Engine.
2. On the Standard bar, click **New Query**.
3. Copy and paste the following example into the query window and click **Execute**.

```





USE AdventureWorks2012;
GO
EXEC sp_control_plan_guide N'DROP ALL';
GO

```

For more information, see [sp_control_plan_guide \(Transact-SQL\)](#).

Enable or Disable a Plan Guide

1/17/2018 • 2 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

You can disable and enable plan guides in SQL Server 2017 by using SQL Server Management Studio or Transact-SQL. Either a single plan guides or all plan guides in a database can be enabled or disabled.

In This Topic

- **Before you begin:**

- [Limitations and Restrictions](#)

- [Security](#)

- **To disable and enable plan guides, using:**

- [SQL Server Management Studio](#)

- [Transact-SQL](#)

Before You Begin

Limitations and Restrictions

- Trying to drop or modify a function, stored procedure, or DML trigger that is referenced by a plan guide, either enabled or disabled, causes an error. Always check for dependencies before dropping or modifying any of the objects listed above.
- Disabling a disabled plan guide or enabling an enabled plan guide has no effect and runs without error.

Security

Permissions

Disabling or enabling an OBJECT plan guide requires ALTER permission on the object (for example: function, stored procedure) that is referenced by the plan guide. All other plan guides require ALTER DATABASE permission.

Using SQL Server Management Studio

To disable or enable a plan guide

1. Click the plus sign to expand the database in which you want to disable or enable a plan guide, and then click the plus sign to expand the **Programmability** folder.
2. Click the plus sign to expand the **Plan Guides** folder.
3. Right-click the plan guide you want to disable or enable and select either **Disable** or **Enable**.
4. In either the **Disable Plan Guide** or **Enable Plan Guide** dialog box, verify that the chosen action was successful and then click **Close**.

To disable or enable all plan guides in a database

1. Click the plus sign to expand the database in which you want to disable or enable a plan guide, and then click the plus sign to expand the **Programmability** folder.
2. Right-click the **Plan Guides** folder and then select either **Enable All** or **Disable All**.

3. In either the **Disable all Plan Guides** or **Enable all Plan Guides** dialog box, verify that the chosen action was successful and then click **Close**.

Using Transact-SQL

To disable or enable a plan guide

1. In **Object Explorer**, connect to an instance of Database Engine.
2. On the Standard bar, click **New Query**.
3. Copy and paste the following example into the query window and click **Execute**.

```
--Create a procedure on which to define the plan guide.
IF OBJECT_ID(N'Sales.GetSalesOrderByCountry', N'P') IS NOT NULL
    DROP PROCEDURE Sales.GetSalesOrderByCountry;
GO
CREATE PROCEDURE Sales.GetSalesOrderByCountry
    (@Country nvarchar(60))
AS
BEGIN
    SELECT *
    FROM Sales.SalesOrderHeader AS h
    INNER JOIN Sales.Customer AS c ON h.CustomerID = c.CustomerID
    INNER JOIN Sales.SalesTerritory AS t ON c.TerritoryID = t.TerritoryID
    WHERE t.CountryRegionCode = @Country;
END
GO
--Create the plan guide.
EXEC sp_create_plan_guide N'Guide3',
    N'SELECT *
    FROM Sales.SalesOrderHeader AS h
    INNER JOIN Sales.Customer AS c ON h.CustomerID = c.CustomerID
    INNER JOIN Sales.SalesTerritory AS t ON c.TerritoryID = t.TerritoryID
    WHERE t.CountryRegionCode = @Country',
    N'OBJECT',
    N'Sales.GetSalesOrderByCountry',
    NULL,
    N'OPTION (OPTIMIZE FOR (@Country = N''US''))';
--Disable the plan guide.
EXEC sp_control_plan_guide N'DISABLE', N'Guide3';
GO
--Enable the plan guide.
EXEC sp_control_plan_guide N'ENABLE', N'Guide3';
GO
```

To disable or enable all plan guides in a database





1. In **Object Explorer**, connect to an instance of Database Engine.
2. On the Standard bar, click **New Query**.
3. Copy and paste the following example into the query window and click **Execute**.

```
--Disable all plan guides in the database.
EXEC sp_control_plan_guide N'DISABLE ALL';
GO
--Enable all plan guides in the database.
EXEC sp_control_plan_guide N'ENABLE ALL';
GO
```

For more information, see [sp_control_plan_guide \(Transact-SQL\)](#).

Monitor Resource Usage (System Monitor)

1/17/2018 • 3 min to read • [Edit Online](#)

THIS TOPIC APPLIES TO:  SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

If you are running Microsoft Windows server operating system, use the System Monitor graphical tool to measure the performance of SQL Server. You can view SQL Server objects, performance counters, and the behavior of other objects, such as processors, memory, cache, threads, and processes. Each of these objects has an associated set of counters that measure device usage, queue lengths, delays, and other indicators of throughput and internal congestion.

NOTE

System Monitor replaced Performance Monitor after Windows NT 4.0.

Benefits of System Monitor

System Monitor can be useful to monitor Windows operating system and SQL Server counters at the same time to determine any correlation between the performance of SQL Server and Windows. For example, monitoring the Windows disk input/output (I/O) counters and the SQL Server Buffer Manager counters at the same time can reveal the behavior of the entire system.

System Monitor allows you to obtain statistics on current SQL Server activity and performance. Using System Monitor, you can:

- View data simultaneously from any number of computers.
- View and change charts to reflect current activity, and show counter values that are updated at a frequency that the user defines.
- Export data from charts, logs, alert logs, and reports to spreadsheet or database applications for further manipulation and printing.
- Add system alerts that list an event in the alert log and can notify you by issuing a network alert.
- Run a predefined application the first time or every time a counter value goes over or under a user-defined value.
- Create log files that contain data about various objects from different computers.
- Append to one file selected sections from other existing log files to form a long-term archive.
- View current-activity reports, or create reports from existing log files.
- Save individual chart, alert, log, or report settings, or the entire workspace setup for reuse.

NOTE

System Monitor replaced the Performance Monitor after Windows NT 4.0. You can use either the System Monitor or Performance Monitor to do these tasks.

System Monitor Performance

When you monitor SQL Server and the Microsoft Windows operating system to investigate performance-related issues, concentrate your initial efforts in three main areas:

- Disk activity
- Processor utilization
- Memory usage

Monitoring a computer on which System Monitor is running can affect computer performance slightly. Therefore, either log the System Monitor data to another disk (or computer) so that it reduces the effect on the computer being monitored, or run System Monitor from a remote computer. Monitor only the counters in which you are interested. If you monitor too many counters, resource usage overhead is added to the monitoring process and affects the performance of the computer that is being monitored.

System Monitor Tasks

TASK DESCRIPTION	TOPIC
Describes when to use System Monitor and discusses performance overhead when you use System Monitor.	Run System Monitor
Describes how to monitor disk counters to determine disk activity and the amount of I/O generated by their SQL Server components.	Monitor Disk Usage
Describes how to monitor an instance of Microsoft SQL Server to determine whether CPU usage rates are within normal ranges.	Monitor CPU Usage
Describes how to monitor an instance of SQL Server to confirm that memory usage is within typical ranges.	Monitor Memory Usage
Describes how to create an alert that is raised when a threshold value for a System Monitor counter has been reached.	Create a SQL Server Database Alert
Describes how to you create charts, alerts, logs, and reports to monitor an instance of SQL Server.	Create Charts, Alerts, Logs, and Reports
Lists objects and counters that System Monitor uses to monitor activity in computers running an instance of SQL Server.	Use SQL Server Objects
Lists objects and counters that System Monitor uses to monitor In-Memory OLTP activity.	SQL Server XTP (In-Memory OLTP) Performance Counters