

Explore query performance optimization

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/explore-query-performance-optimization/1-introduction>

Introduction

Completed

One of the most crucial skills to acquire in database performance tuning is the ability to read and understand query execution plans. These plans provide detailed insights into the behavior of the database engine as it executes queries and retrieves results. By analyzing execution plans, you can identify inefficiencies, optimize query performance, and ensure that your database runs smoothly.

The Query Store is an invaluable tool for quickly identifying your most expensive queries and tracking changes in performance over time. It offers comprehensive data collection, including automated query plan and execution runtime analysis. This allows you to pinpoint performance bottlenecks and make informed decisions to improve query efficiency.

SQL Server implements locking and blocking mechanisms to manage concurrency and ensure data consistency. These mechanisms prevent conflicts and maintain the integrity of your data when multiple users access the database simultaneously. Also, you can adjust isolation levels in SQL Server to fine-tune concurrency management. By selecting the appropriate isolation level, you can balance the trade-offs between data consistency and performance, ensuring that your database operates optimally under various workloads.

2. Understand query plans

<https://learn.microsoft.com/en-us/training/modules/explore-query-performance-optimization/2-understand-query-plans>

Understand query plans

Completed

Understanding how database optimizers work is essential before diving into execution plan details. SQL Server uses a cost-based query optimizer, which calculates the cost for multiple possible plans based on statistics it has on the columns being used and the potential indexes for each operation in the query plan. This information helps the optimizer determine the total cost for each plan. Complex queries can have thousands of possible execution plans, but the optimizer doesn't evaluate every single one. Instead, it uses heuristics to identify plans that are likely to perform well and then selects the lowest cost plan from those evaluated.

Since the query optimizer is cost-based, it's crucial to provide it with accurate inputs for decision-making. SQL Server relies on statistics to track the distribution of data in columns and indexes, and these statistics must be kept up to date to avoid generating suboptimal execution plans. Although SQL Server automatically updates its statistics as data changes in a table, more frequent updates may be necessary for rapidly changing data. The optimizer considers many factors when building a plan, including the database's compatibility level, row estimates based on statistics, and available indexes.

When a user submits a query to the database engine, the following process happens:

1. The query is parsed for proper syntax, and if correct, a parse tree of database objects is generated.
2. The parse tree is then input to a database engine component called the *Algebrizer* for binding. This step validates that columns and objects in the query exist and identifies the data types being processed. The output is a query processor tree, which serves as input for the next step.
3. Query optimization is CPU-intensive, so the database engine caches execution plans in a special memory area called the plan cache. If a plan for the query already exists, it's retrieved from the cache. Each query in the cache has a hash value generated based on the T-SQL in the query, known as the *query_hash*. The engine generates a *query_hash* for the current query and checks for matches in the plan cache.
4. If no plan exists, the Query Optimizer uses its cost-based optimizer to generate several execution plan options based on statistics about the columns, tables, and indexes used in the query. The output is a query execution plan.
5. The query is executed using an execution plan from the plan cache or a new plan generated in the previous step. The output is the results of your query.

Note

To learn more about how the query processor works, see [Query Processing Architecture Guide](#)

Let's look at an example. Consider the following query:

```
SELECT orderdate,  
       AVG(salesAmount)  
FROM FactResellerSales  
WHERE ShipDate = '2013-07-07'  
GROUP BY orderdate;
```

In this example, SQL Server checks for the existence of the *OrderDate*, *ShipDate*, and *SalesAmount* columns in the *FactResellerSales* table. If these columns exist, SQL Server generates a hash value for the query and examines the plan cache for a matching hash value. If a matching hash value is found, the engine attempts to reuse the plan. If no matching hash value is found, SQL Server examines the available statistics on the *OrderDate* and *ShipDate* columns.

The `WHERE` clause referencing the *ShipDate* column is known as the predicate in this query. If there's a nonclustered index that includes the *ShipDate* column, SQL Server will likely include it in the plan, provided the costs are lower than retrieving data from the clustered index. The optimizer then chooses the lowest cost plan from the available options and executes the query.

Query plans combine a series of relational operators to retrieve data and capture information such as estimated row counts. Another element of the execution plan is the memory required for operations like joining or sorting data, known as the memory grant. The memory grant highlights the importance of statistics. If SQL Server estimates an operator returns 10,000,000 rows when it actually returns 100, a larger memory grant is allocated to the query. An excessively large memory grant can cause two issues. First, the query may encounter a `RESOURCE_SEMAPHORE` wait, indicating it's waiting for SQL Server to allocate a large amount of memory. SQL Server defaults to waiting for 25 times the cost of the query (in seconds) before executing, up to 24 hours. Second, if there isn't enough memory available when the query is executed, it spills to tempdb, which is slower than operating in memory.

The execution plan also stores other metadata about the query, such as the database compatibility level, the degree of parallelism, and the parameters supplied if the query is parameterized.

Query plans can be viewed in either a graphical representation or a text-based format. Text-based options are invoked with `SET` commands and apply only to the current connection. These plans can be viewed anywhere you can run T-SQL queries.

Most DBAs prefer graphical plans because they allow you to see the plan as a whole, including the *shape* of the plan. There are several ways to view and save graphical query plans. The most common tool for this purpose is SQL Server Management Studio. Additionally, there are third-party tools that support viewing graphical execution plans.

There are three different types of execution plans.

Estimated Execution Plan

This type of execution plan is generated by the query optimizer. The metadata and size of the query memory grant are based on estimates from the statistics present in the database at the time of query compilation. To see a text-based estimated plan, run the command `SET SHOWPLAN_ALL ON` before executing the query. When you run the query, you see the steps of the execution plan, but the query won't be executed, and you won't see any results. The `SET` option remains in effect until you set it `OFF`.

Actual Execution Plan

This type of plan is the same as the estimated plan; however, it also includes the execution context for the query. This context contains the estimated and actual row counts, any execution warnings, the actual degree of parallelism (number of processors used), and the elapsed and CPU times used during execution. To see a text-based actual plan, run the command `SET STATISTICS PROFILE ON` before executing the query. The query executes, and you get both the plan and the results.

Live Query Statistics

This plan viewing option combines the estimated and actual plans into an animated plan that displays execution progress through the operators. It refreshes every second and shows the actual number of rows flowing through the operators. Another benefit of Live Query Statistics is that it shows the handoff from operator to operator, which can be helpful in troubleshooting performance issues. Because this type of plan is animated, it's only available as a graphical plan.

3. Explain estimated and actual query plans

<https://learn.microsoft.com/en-us/training/modules/explore-query-performance-optimization/3-explain-estimated-actual>

Explain estimated and actual query plans

Completed

Actual versus estimated execution plans can be confusing. The difference is that the actual plan includes runtime statistics that aren't captured in the estimated plan. The operators used and the order of execution will be the same as the estimated plan in nearly all cases. Another consideration is that capturing an actual execution plan requires the query to be executed, which can be time-consuming or not possible. For example, an `UPDATE` statement can only be run once. However, if you need to see query results and the plan, you need to use one of the actual plan options.

SQLQuery15.sql - (local).AdventureWorks2017 (VM1\jdantoni (76))* - Microsoft SQL Server Management Studio (Administrator)

File Edit View Query Project Tools Window Help

AdventureWorks2017 Execute

SQLQuery15.sql...jdantoni (76))* Top Source...tureWorks2017 ExecuteRandom...jdantoni (76)

Estimated Execution Plan

Actual Execution Plan

```

DECLARE @SalesPersonID INT;
SELECT SalesPersonID, OrderDate
from Sales.SalesOrderHeader
WHERE SalesPersonID= @SalesPersonID
OPTION (OPTIMIZE FOR (@SalesPersonID = 288));

```

100 %

Results Messages Execution plan

Query 1: Query cost (relative to the batch): 100%

SELECT SalesOrderId, OrderDate from Sales.SalesOrderHeader WHERE SalesPersonID= @SalesPersonID OPT

Nested Loops (Inner Join)

Cost: 0 %

0.001s

473 of 130 (363%)

Index Seek (NonClustered)

[SalesOrderHeader].[IX_SalesOrderHeader_...]

Cost: 1 %

0.000s

473 of 130 (363%)

Key Lookup (Clustered)

[SalesOrderHeader].[PK_SalesOrderHeader_...]

Cost: 99 %

0.000s

473 of 130 (363%)

As shown, you can generate an estimated plan in SSMS by selecting the button indicated by the estimated query plan box (or using the keyboard command **Control+L**). You can generate the actual plan by selecting the icon shown (or using the keyboard command **Control+M**), and then executing the query. The two option buttons work differently. The *Include Estimated Query Plan* button responds immediately to any query highlighted (or the entire workspace, if nothing is highlighted), whereas the *Include Actual Query Plan* button requires the query to be executed.

There's overhead to both executing a query and generating an estimated execution plan, so viewing execution plans should be done carefully in a production environment.

Typically, you can use the estimated execution plan while writing your query to understand its performance characteristics, identify missing indexes, or detect query anomalies. The actual execution plan is best used to understand the runtime performance of the query and, most importantly, gaps in statistical data that cause the query optimizer to make suboptimal choices based on the data it has available.

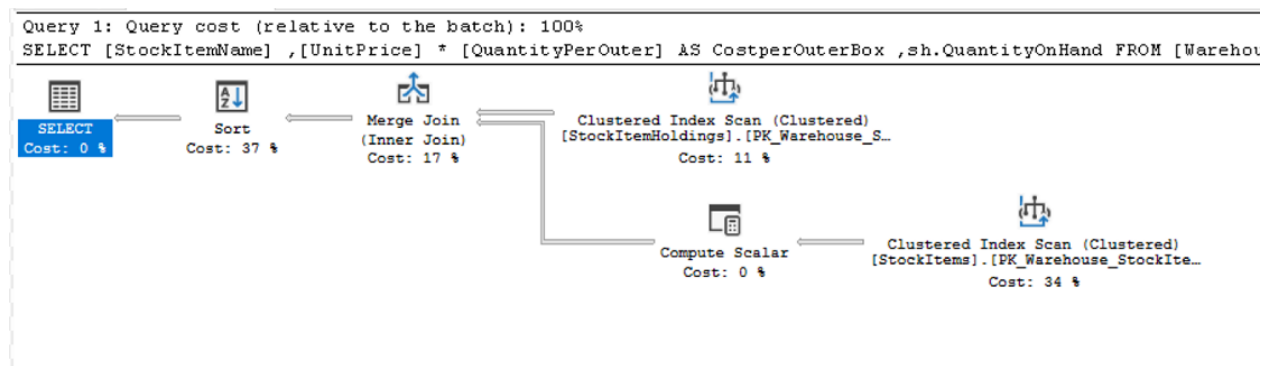
Read a query plan

Execution plans show you what tasks the database engine is performing while retrieving the data needed to satisfy a query. Let's dive into the plan.

```
SELECT [stockItemName]
,[UnitPrice] * [QuantityPerOuter] AS CostPerOuterBox
,[QuantityOnHand]

FROM [Warehouse].[StockItems] s
JOIN [Warehouse].[StockItemHoldings] sh ON s.StockItemID = sh.StockItemID
ORDER BY CostPerOuterBox;
```

This query is joining the *StockItems* table to the *StockItemHoldings* table where the values in the column *StockItemID* are equal. The database engine has to first identify those rows before it can process the rest of the query.



Each icon in the plan represents a specific operation, which corresponds to the various actions and decisions that make up an execution plan. The SQL Server database engine has over 100 query operators that can be part of an execution plan. Under each operator icon, there's a cost percentage relative to the total cost of the query. Even an operation showing a cost of 0% still represents some cost. In fact, 0% is due to rounding, as graphical plan costs are always shown as whole numbers, and the real percentage is something less than 0.5%.

The flow of execution in an execution plan is from right to left, and top to bottom, so in this plan, the Clustered Index Scan operation on the *StockItemHoldings.PK_Warehouse_StockItemHoldings* clustered index is the first operation in the query. The widths of the lines that connect the operators are based on the estimated number of rows of data that flow onward to the next operator. A thick arrow is an indicator of large operator to operator transfer and may be indicative of an opportunity to tune a query. You can also hold your mouse over an operator and see additional information in a ToolTip.



Clustered Index Scan (Clustered)
[StockItems].[PK_Warehouse_StockItem...]
Cost: 34 %

Clustered Index Scan (Clustered)	
Scanning a clustered index, entirely or only a range.	
Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Estimated Execution Mode	Row
Storage	RowStore
Estimated Operator Cost	0.0131613 (34%)
Estimated I/O Cost	0.0127546
Estimated Subtree Cost	0.0131613
Estimated CPU Cost	0.0004067
Estimated Number of Executions	1
Estimated Number of Rows	227
Estimated Number of Rows to be Read	227
Estimated Row Size	128 B
Ordered	True
Node ID	4
Object [WideWorldImporters].[Warehouse].[StockItems]. [PK_Warehouse_StockItems] [s]	
Output List [WideWorldImporters].[Warehouse].[StockItems].StockItemID, [WideWorldImporters].[Warehouse].[StockItems].StockItemName, [WideWorldImporters].[Warehouse]. [StockItems].QuantityPerOuter, [WideWorldImporters]. [Warehouse].[StockItems].UnitPrice	

The tooltip highlights the cost and estimates for the estimated plan, and for an actual plan, it includes comparisons to the actual rows and costs. Each operator also has properties that provide more details than the tooltip. By right-clicking on a specific operator, you can select the Properties option from the context menu to see the full property list. This option opens a separate Properties pane in SQL Server Management Studio, which by default is on the right side. Once the Properties pane is open, selecting any operator populates the **Properties** list with details for that operator. Alternatively, you can open the Properties pane by selecting on **View** in the main SQL Server Management Studio menu and choosing **Properties**.

Properties	
Clustered Index Scan (Clustered)	
Misc	
Defined Values	[WideWorldImporters].[Warehouse].[StockItems].Stc...
Description	Scanning a clustered index, entirely or only a range.
Estimated CPU Cost	0.0004067
Estimated Execution Mode	Row
Estimated I/O Cost	0.0127546
Estimated Number of Executions	1
Estimated Number of Rows	227
Estimated Number of Rows to be Read	227
Estimated Operator Cost	0.0131613 (34%)
Estimated Rebinds	0
Estimated Rewinds	0
Estimated Row Size	128 B
Estimated Subtree Cost	0.0131613
Forced Index	False
ForceScan	False
ForceSeek	False
Logical Operation	Clustered Index Scan
Node ID	4
NoExpandHint	False
Object	[WideWorldImporters].[Warehouse].[StockItems].[PK_V
Alias	[s]
Database	[WideWorldImporters]
Index	[PK_Warehouse_StockItems]
Index Kind	Clustered
Schema	[Warehouse]
Storage	RowStore
Table	[StockItems]
Ordered	True
Output List	[WideWorldImporters].[Warehouse].[StockItems].Stock
[1]	[WideWorldImporters].[Warehouse].[StockItems].Stock
[2]	[WideWorldImporters].[Warehouse].[StockItems].Stock
[3]	[WideWorldImporters].[Warehouse].[StockItems].Quan
[4]	[WideWorldImporters].[Warehouse].[StockItems].UnitP
Parallel	False
Physical Operation	Clustered Index Scan
Scan Direction	FORWARD
Storage	RowStore
TableCardinality	227

The Properties pane includes additional information and shows the output list, detailing the columns being passed to the next operator. These columns may indicate that a nonclustered index is needed to improve query performance when analyzed with a clustered index scan. Since a clustered index scan operation reads the entire table, a nonclustered index on the *StockItemID* column in each table could be more efficient in this scenario.

Lightweight query profiling

When you generate actual execution plans, whether using SSMS or the Extended Events monitoring infrastructure, it can introduce significant overhead. Therefore, this process is typically reserved for live site troubleshooting efforts. Observer overhead, as it's known, is the cost of monitoring a running application. In some scenarios, this cost can be just a few percentage points of CPU utilization, but in other cases, like capturing actual execution plans, it can significantly slow down individual query performance. The legacy profiling in SQL Server's engine could produce up to 75% overhead for capturing query information, whereas the lightweight profiling has a maximum overhead of around 2%.

In the first version of lightweight profiling, it collected row count and I/O utilization information (the number of logical and physical reads and writes performed by the database engine to satisfy a given query). Additionally, a new extended event called *query_thread_profile* was introduced to allow data from each operator in a query plan to be inspected. In the initial version of lightweight profiling, using the feature requires trace flag 7412 to be enabled globally.

If lightweight profiling isn't enabled globally, you can use the `USE HINT` query hint with `QUERY_PLAN_PROFILE` to enable lightweight profiling at the query level. When a query with this hint completes execution, a *query_plan_profile* extended event is generated, providing an actual execution plan. Here's an example of a query with this hint:

```
SELECT [stockItemName]
, [UnitPrice] * [QuantityPerOuter] AS CostPerOuterBox
, [ QuantityonHand]
FROM [Warehouse].[StockItems] s
    JOIN [Warehouse].[StockItems] sh ON s.StockItemID = sh.StockItemID
ORDER BY CostPerOuterBox
OPTION(USE HINT ('QUERY_PLAN_PROFILE'));
```

Last query plans stats

Lightweight profiling is enabled by default in both SQL Server 2019 and Azure SQL Database and managed instance. Lightweight profiling is also available as a database scoped configuration option, called `LIGHTWEIGHT_QUERY_PROFILING`. With the database scoped option, you can disable the feature for any of your user databases independent of each other.

Also, there's a dynamic management function called `sys.dm_exec_query_plan_stats`, which can show you the last known actual query execution plan for a given plan handle. In order to see the last known actual query plan through the function, you can enable trace flag 2451 server-wide.

Alternatively, you can enable this functionality using a database scoped configuration option called `LAST_QUERY_PLAN_STATS`.

You can combine this function with other objects to get the last execution plan for all cached queries:

```

SELECT *
FROM sys.dm_exec_cached_plans AS cp
      CROSS APPLY sys.dm_exec_sql_text(plan_handle) AS st
      CROSS APPLY sys.dm_exec_query_plan_stats(plan_handle) AS qps;
GO

```

This functionality lets you quickly identify the runtime stats for the last execution of any query in your system, with minimal overhead. The following image shows how to retrieve the plan. If you select the execution plan XML, which will be the first column of results, it displays the execution plan shown in the second image below.

SQLQuery1.sql - VM...VM1\jdantoni (88)*

```

DBCC TRACEON (2451, -1)
GO
USE WideWorldImporters
GO

SELECT ol.StockItemID, [Description], SUM(Quantity - PickedQuantity) AS AllocatedQuantity
FROM Sales.OrderLines AS ol WITH (NOLOCK)
GROUP BY ol.StockItemID, [Description];
GO

SELECT qps.query_plan
      ,st.TEXT
      ,DB_NAME(st.dbid) DBName
      ,OBJECT_NAME(st.objectid) ObjectName
      ,cp.usecounts
      ,cp.objtype
FROM sys.dm_exec_cached_plans AS cp
CROSS APPLY sys.dm_exec_sql_text(plan_handle) AS st
CROSS APPLY sys.dm_exec_query_plan_stats(plan_handle) AS qps
WHERE st.encrypted = 0 and st.text like 'SELECT ol.%'

```

	query_plan	TEXT	DBName	ObjectName	usecounts	objtype
1	<ShowPlanXML xmlns="http://schemas.microsoft.com...">	SELECT ol.StockItemID, [Description], SUM(Quant...	WideWorldImporters	NULL	1	Adhoc

As you can see from the properties of the *Columnstore Index Scan* in the following image, the plan retrieved from the cache has actual number of rows retrieved in the query.

ExecutionPlan1.sqlplan* SQLQuery1.sql - VM...VM1\jdantoni (88)*

Query 1: Query cost (relative to the batch): 100%

```

SELECT ol.StockItemID, [Description], SUM(Quantity - PickedQuantity) AS AllocatedQuantity FROM Sales.OrderLines AS ol WITH (NOLOCK) GROUP BY ol.StockItemID, [Description]

```

Hash Match (Aggregate)
Cost: 0 %
227 of 50963 (0%)

Compute Scalar
Cost: 0 %
157422 of 231412 (68%)

Columnstore Index Scan (NonClustered)
[OrderLines].[NCCX_Sales_OrderLines...]
Cost: 0 %
231412

Columnstore Index Scan (NonClustered)
23: Scan a columnstore index, entirely or only a range.

Physical Operation	Columnstore Index Scan
Logical Operation	Index Scan
Actual Execution Mode	Batch
Estimated Execution Mode	Batch
Storage	ColumnStore
Actual Number of Rows	157422
Actual Number of Batches	1509
Estimated I/O Cost	0.003125
Estimated Operator Cost	0.028596 (5%)
Estimated CPU Cost	0.025471
Estimated Subtree Cost	0.028596
Number of Executions	1
Estimated Number of Executions	1
Estimated Number of Rows	231412
Estimated Number of Rows to be Read	231412
Estimated Row Size	118 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	4
Object	[WideWorldImporters].[Sales].[OrderLines].[NCCX_Sales_OrderLines]
Output List	[WideWorldImporters].[Sales].[OrderLines].OrderLineID, [WideWorldImporters].[Sales].[OrderLines].StockItemID, [WideWorldImporters].[Sales].[OrderLines].Description, [WideWorldImporters].[Sales].[OrderLines].Quantity, [WideWorldImporters].[Sales].[OrderLines].PickedQuantity, Generation1004

4. Describe dynamic management views and functions

<https://learn.microsoft.com/en-us/training/modules/explore-query-performance-optimization/4-describe-dynamic-management-views-functions>

Describe dynamic management views and functions

Completed

SQL Server provides several hundred dynamic management objects. These objects contain system information that can be used to monitor the health of a server instance, diagnose problems, and tune performance. Dynamic management views and functions return internal data about the state of the database or the instance. Dynamic Management Objects can be either views (DMVs) or functions (DMFs), but most people use the acronym DMV to refer to both types of object.

There are two levels of DMVs, server scoped and database scoped.

- **Server scoped objects** – require `VIEW SERVER STATE` permission on the server
- **Database scoped objects** – require the `VIEW DATABASE STATE` permission within the database

The names of the DMVs are all prefixed with **sys.dm_** followed by the functional area and then the specific function of the object. SQL Server supports three categories of DMVs:

- Database-related dynamic management objects
- Query execution related dynamic management objects
- Transaction related dynamic management objects

To learn about queries to monitor server and database performance, see [Monitoring Microsoft Azure SQL Database and Azure SQL Managed Instance performance using dynamic management views](#).

Note

For older versions of SQL Server where the query store isn't available, you can use the view `sys.dm_exec_cached_plans` with the functions `sys.dm_exec_sql_text` and `sys.dm_exec_query_plan` to return information about execution plans. However, unlike with Query Store, you won't be able to see changes in plans for a given query.

Azure SQL Database has a slightly different set of the DMVs available than SQL Server; some objects are available only in Azure SQL Database, while other objects are only available in SQL Server. Some

are scoped at the server level and aren't applicable in the Azure model (the `waits_stats` DMV is an example of a server-scoped DMV), while others are specific to Azure SQL Database, like `sys.dm_db_resource_stats` and provide Azure-specific information that isn't available in (or relevant to) SQL Server.

5. Explore Query Store

<https://learn.microsoft.com/en-us/training/modules/explore-query-performance-optimization/5-explore-query-store>

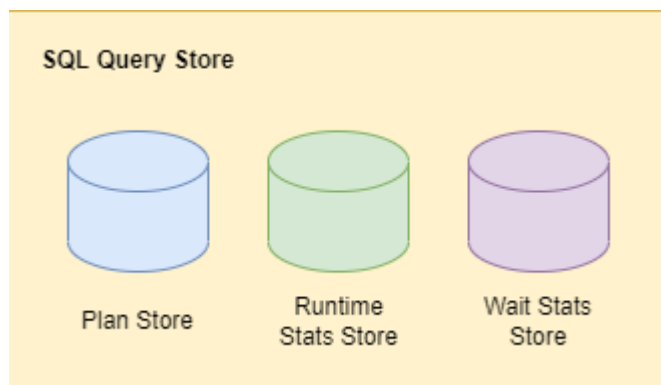
Explore Query Store

Completed

The SQL Server Query Store is a per-database feature that automatically captures a history of queries, plans, and runtime statistics, simplifying performance troubleshooting and query tuning. It also provides insights into database usage patterns and resource consumption.

The Query Store consists of three stores:

- **Plan store:** Stores estimated execution plan information.
- **Runtime stats store:** Stores execution statistics information.
- **Wait stats store:** Persists wait statistics information.



Enable the Query Store

The Query Store is enabled by default in Azure SQL databases. If you want to use it with SQL Server and Azure Synapse Analytics, you need to enable it first. To enable the Query Store feature, use the following query valid for your environment:

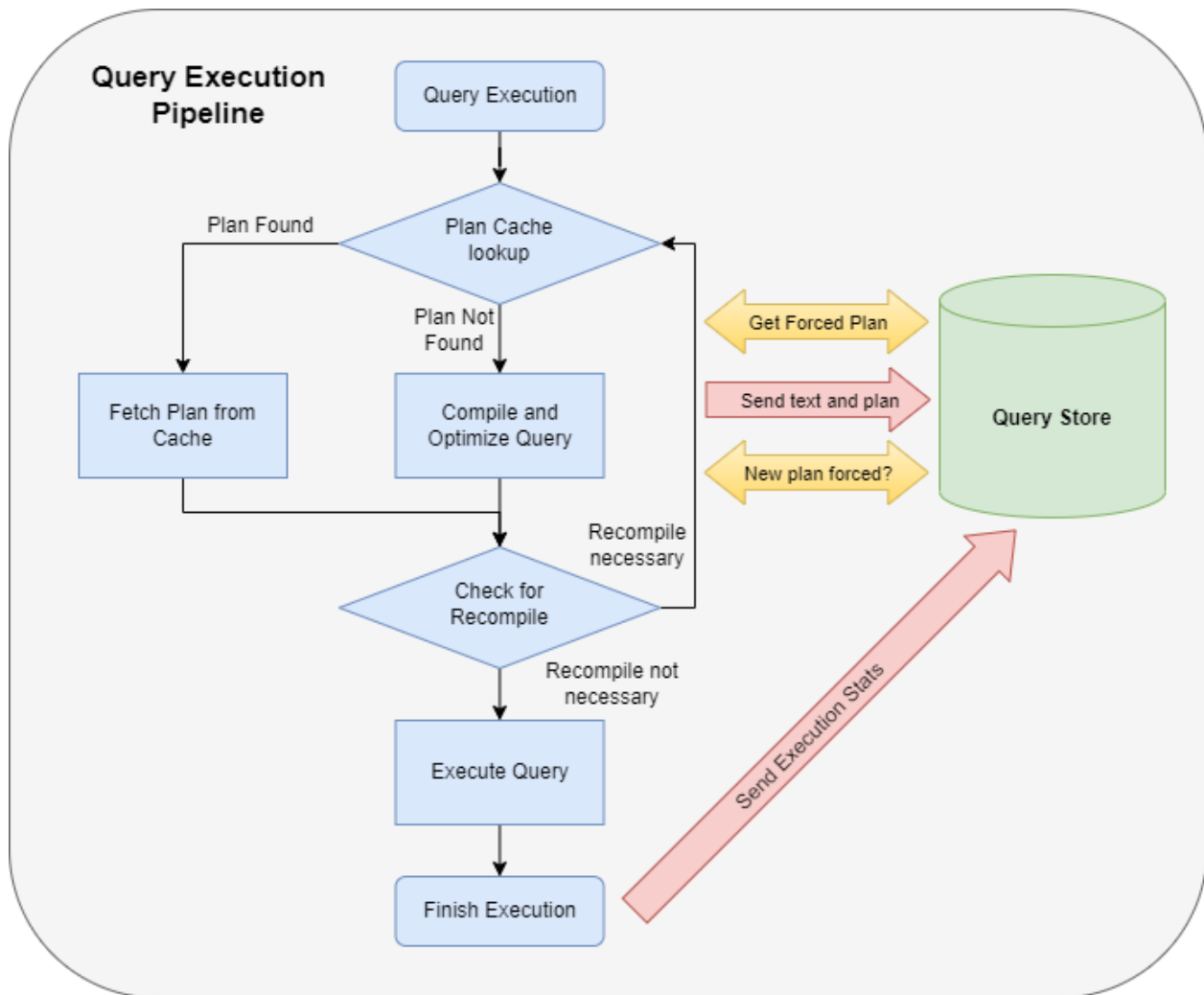
```
-- SQL Server
ALTER DATABASE <database_name> SET QUERY_STORE = ON (OPERATION_MODE = READ_WRITE);

-- Azure Synapse Analytics
ALTER DATABASE <database_name> SET QUERY_STORE = ON;
```

How the Query Store collects data

The Query Store integrates with the query processing pipeline at multiple stages. At each integration point, data is collected in memory and written to disk asynchronously to minimize I/O overhead. The integration points are as follows:

1. When a query executes for the first time, its query text and initial estimated execution plan are sent to the Query Store and persisted.
2. The plan updates in the Query Store when a query recompile. If the recompile results in a newly generated execution plan, it also persists in the Query Store to augment the previous plans. Additionally, the Query Store keeps track of the execution statistics for each query plan for comparison purposes.
3. During the compile and check for recompile phases, the Query Store identifies if there's a forced plan for the query to be executed. The query is recompiled if the Query Store provides a forced plan different from the plan in the procedure cache.
4. When a query executes, its runtime statistics persist in the Query Store. The Query Store aggregates this data to ensure an accurate representation of every query plan.



To learn more about how Query Store collects data, see [How Query Store collects data](#).

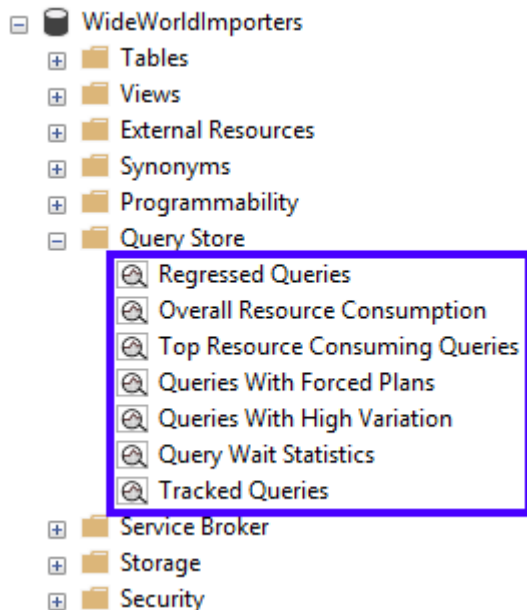
Common scenarios

The SQL Server Query Store provides valuable insights into the performance of database operations. Common scenarios include:

- Identifying and fixing performance regressions due to inferior query execution plan selection.
- Identifying and tuning the highest resource consumption queries.
- A/B testing to evaluate the impacts of database and application changes.
- Ensuring performance stability after SQL Server upgrades.
- Determining the most frequently used queries.
- Auditing the history of query plans for a query.
- Identifying and improving unplanned workloads.
- Understanding the prevalent wait categories of a database and the contributing queries and plans affecting wait times.
- Analyzing database usage patterns over time in terms of resource consumption (CPU, I/O, Memory).

Discover the Query Store views

Once Query Store is enabled on a database, the Query Store folder is visible for the database in Object Explorer. For Azure Synapse Analytics, the Query Store views are displayed under System Views. The Query Store views provide aggregated, quick insights into the performance aspects of the SQL Server database.

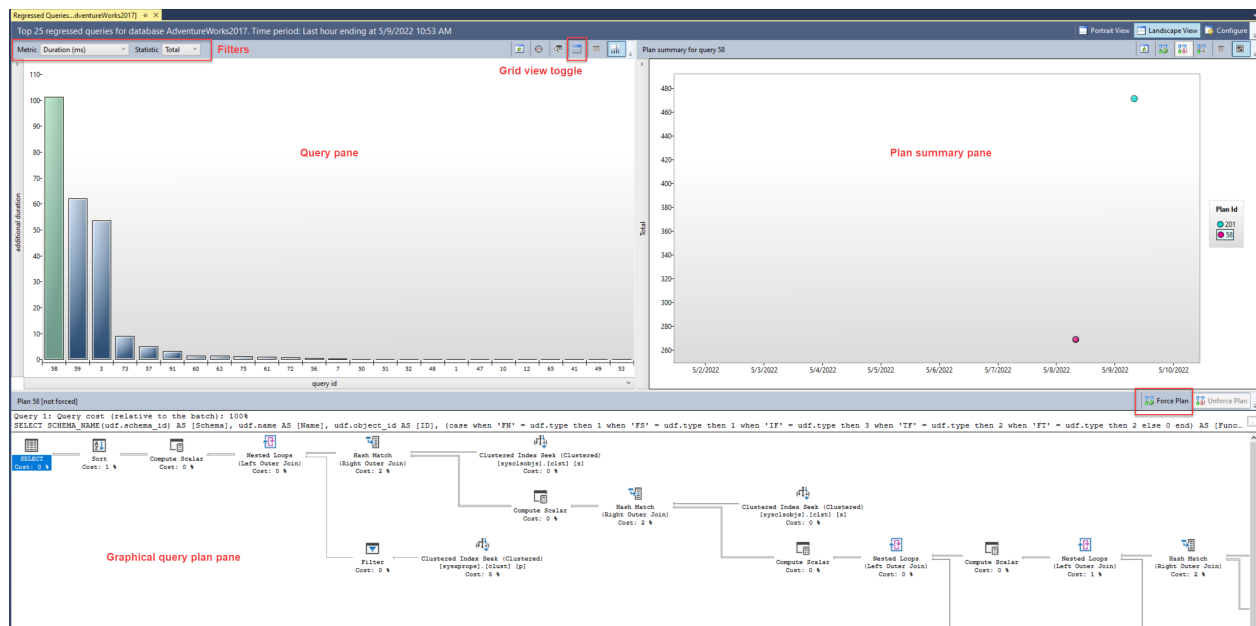


Regressed Queries

A regressed query experiences performance degradation over time due to execution plan changes. Estimated execution plans can change due to various factors, including schema changes, statistics changes, and index changes. Investigating the procedure cache might be the first instinct, but it only stores the latest execution plan for a query, and plans can be evicted based on the system's memory demands. The Query Store, however, persists multiple execution plans for each query, allowing the flexibility to choose a specific plan through *plan forcing* to address query performance regression caused by plan changes.

The **Regressed Queries** view can pinpoint queries whose execution metrics are regressing due to execution plan changes over a specified timeframe. This view allows filtering based on a selected metric (such as duration, CPU time, row count, and more) and a statistic (total, average, min, max, or standard deviation). It then lists the top 25 regressed queries based on the provided filter. By default, a graphical bar chart view of the queries is displayed, but you can optionally view the queries in a grid format.

After selecting a query from the top-left query pane, the plan summary pane displays the persisted query plans associated with the query over time. Selecting a query plan in the Plan Summary pane shows a graphical query plan in the bottom pane. Toolbar buttons in both the plan summary pane and graphical query plan pane allow you to force the selected plan for the selected query. This pane structure and behavior are consistently used across all SQL Query views.

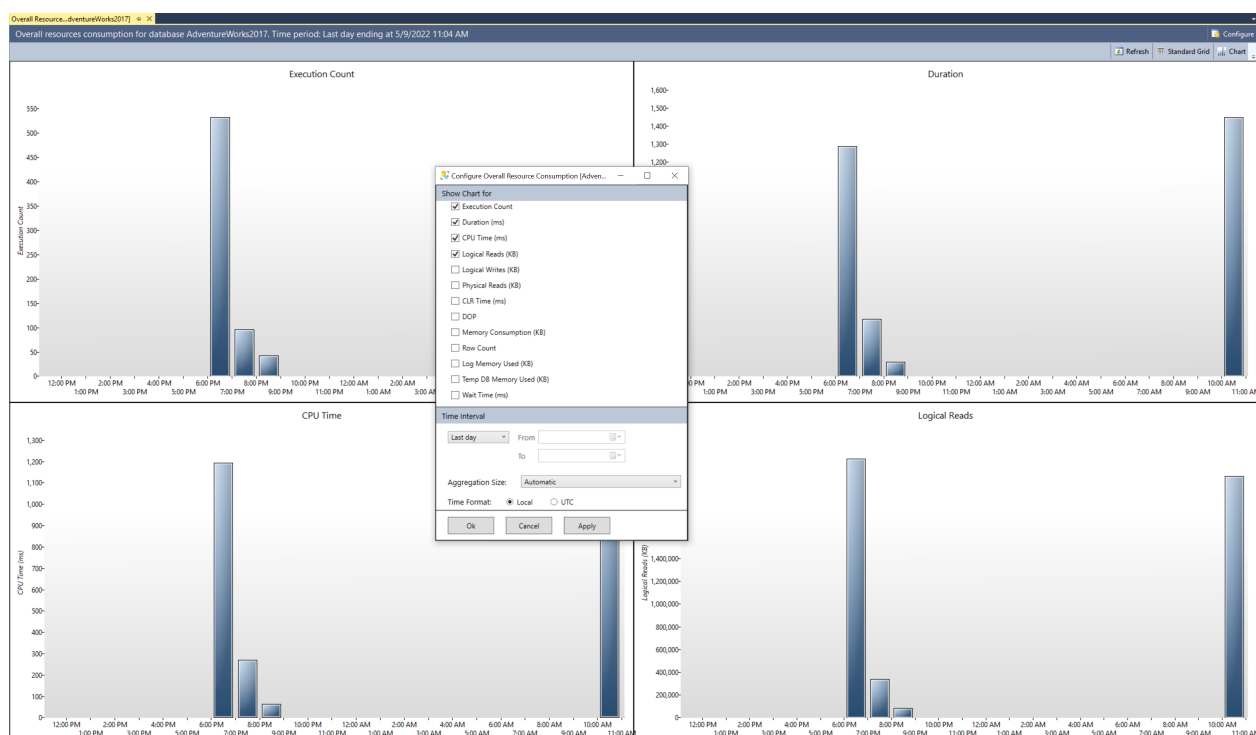


Alternatively, you can use the `sp_query_store_force_plan` stored procedure to use plan forcing.

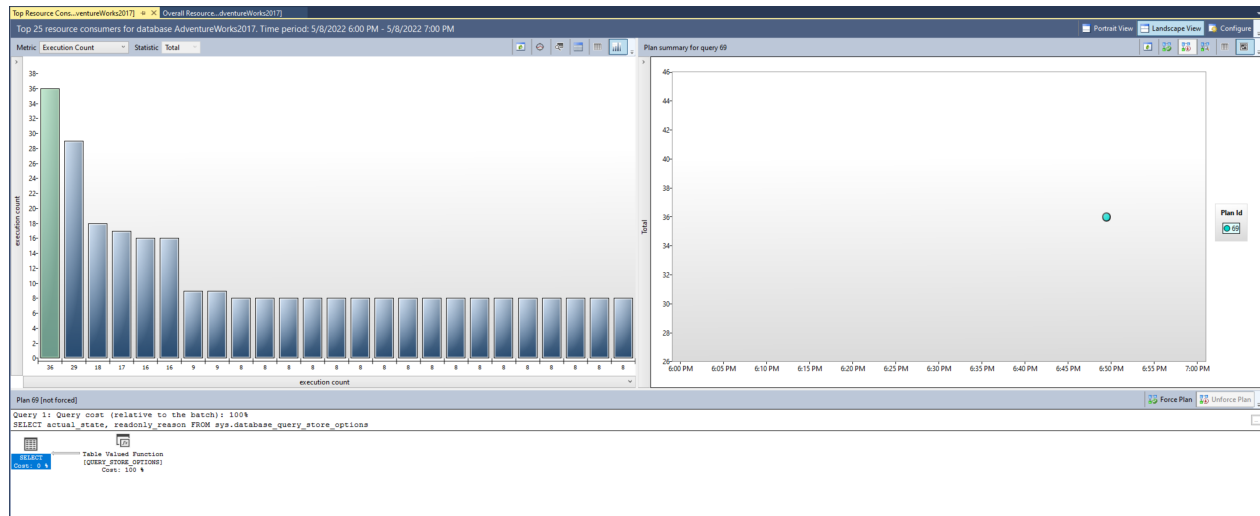
```
EXEC sp_query_store_force_plan @query_id=73, @plan_id=79
```

Overall Resource Consumption

The **Overall Resource Consumption** view allows for analyzing total resource consumption for multiple execution metrics (such as execution count, duration, wait time, and more) for a specified timeframe. The rendered charts are interactive; when selecting a measure from one of the charts, a drill through view displaying the queries associated with the chosen measure displays in a new tab.

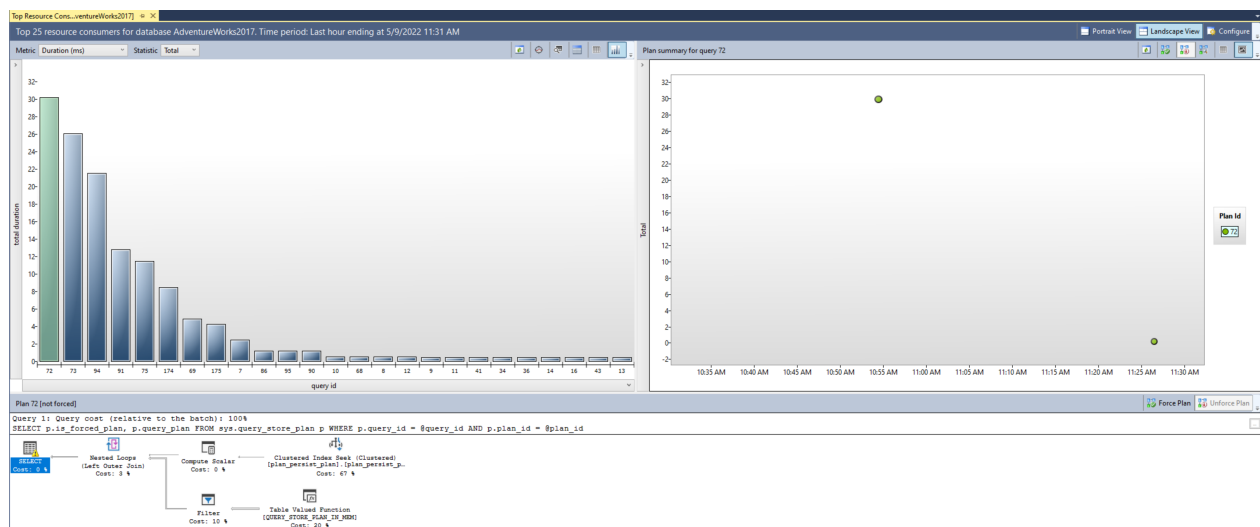


The details view provides the top 25 resource consumer queries that contributed to the metric that was selected. This details view uses the consistent interface that allows for the inspection of the associated queries and their details, evaluate saved estimated query plans, and optionally use plan forcing to improve performance. This view is valuable when system resource contention becomes an issue, such as when CPU usage reaches capacity.

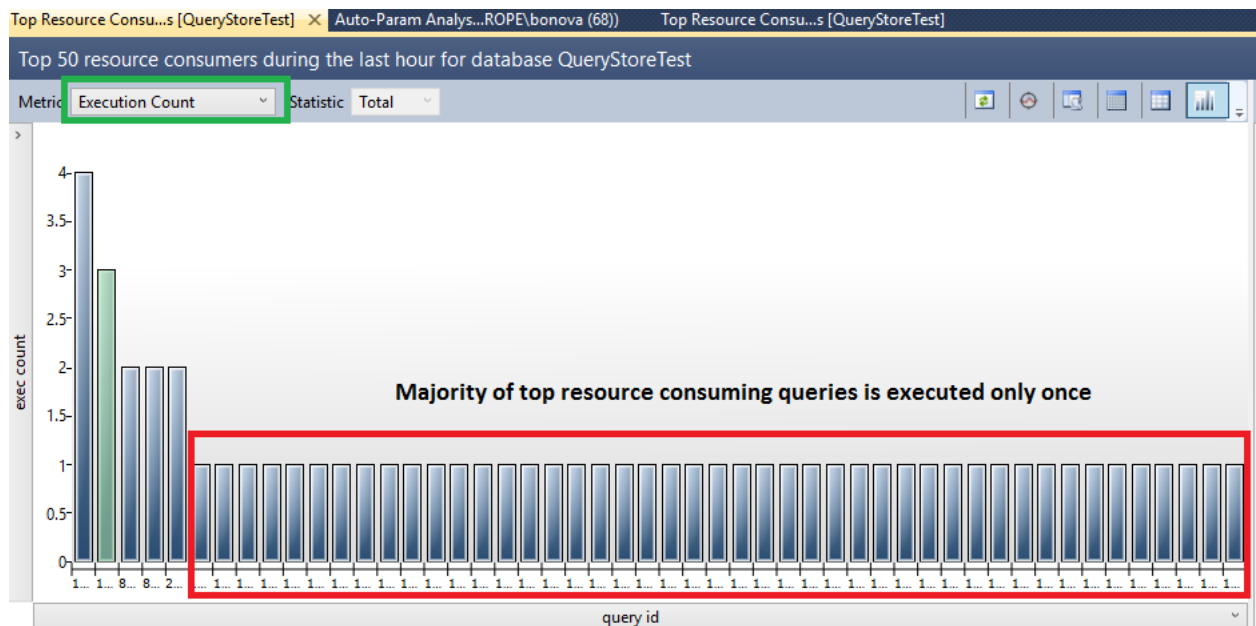


Top Resource Consuming Queries

The **Top Resource Consuming Queries** view is similar to the details drill down of the Overall Resource Consumption view. It also allows for selecting a metric and a statistic as a filter. However, the queries it displays are the top 25 most impactful queries based on the chosen filter and timeframe.

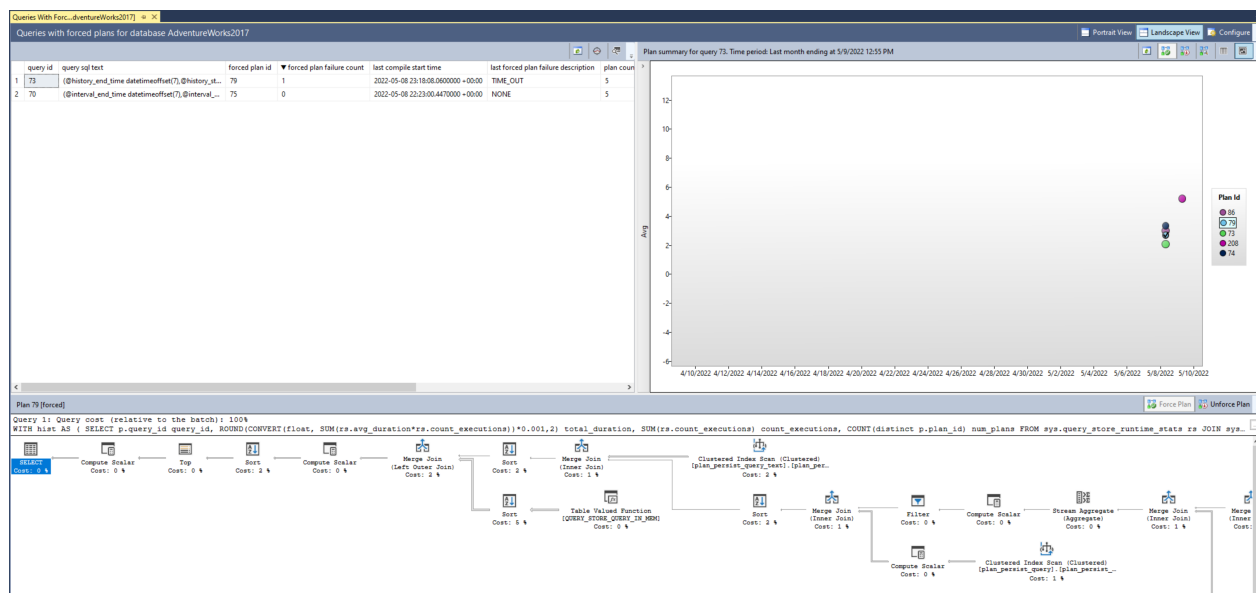


The **Top Resource Consuming Queries** view provides the first indication of the unplanned nature of the workload when identifying and improving unplanned workloads. For example, in the following image, the *Execution Count* metric and the *Total* statistic are selected to unveil that approximately 90% of the top resource-consuming queries are only executed once.



Queries With Forced Plans

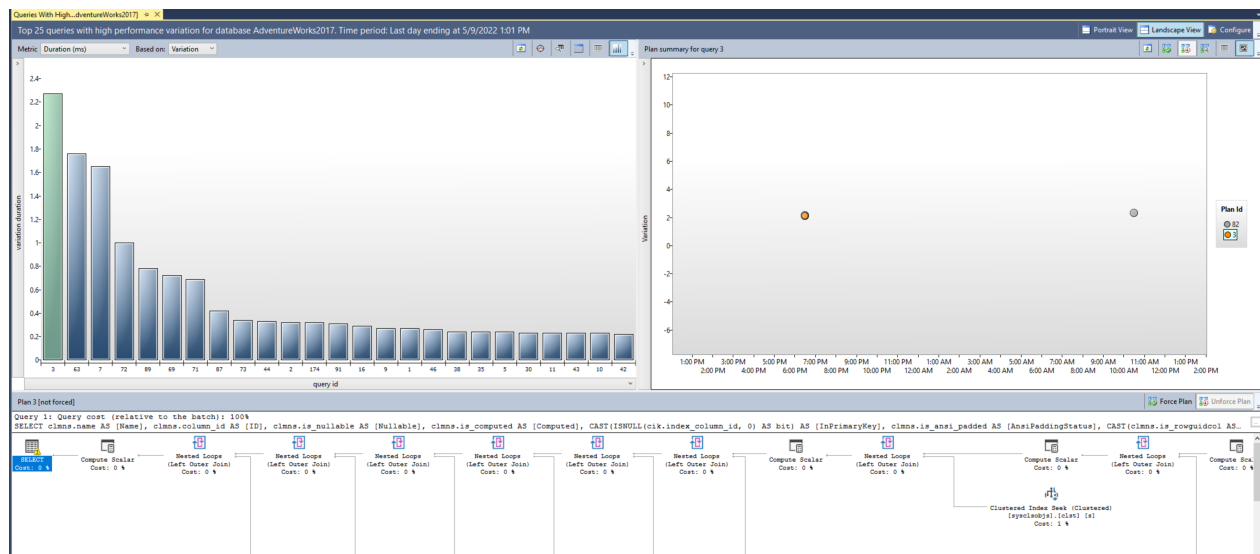
The **Queries With Forced Plans** view provides a quick look into the queries that have forced query plans. This view becomes relevant if a forced plan no longer performs as expected and needs to be reevaluated. This view provides the ability to review all persisted estimated execution plans for a selected query easily determining if another plan is now better suited for performance. If so, toolbar buttons are available to unforce a plan as required.



Queries With High Variation

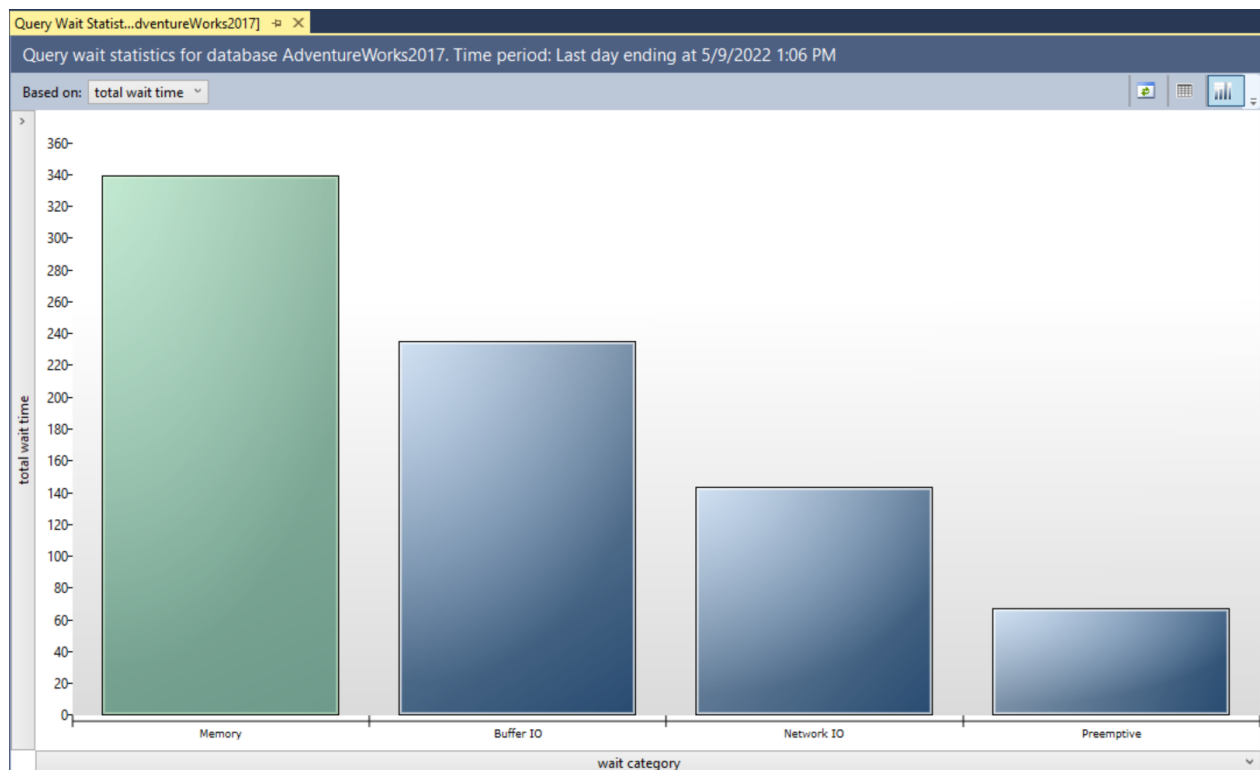
Query performance can vary between executions. The **Queries with High Variation** view contains an analysis of queries that have the highest variation or standard deviation for a selected metric. The interface is consistent with most Query Store views allowing for query detail inspection, execution

plan evaluation, and optionally forcing a specific plan. Use this view to tune unpredictable queries into a more consistent performance pattern.



Query Wait Statistics

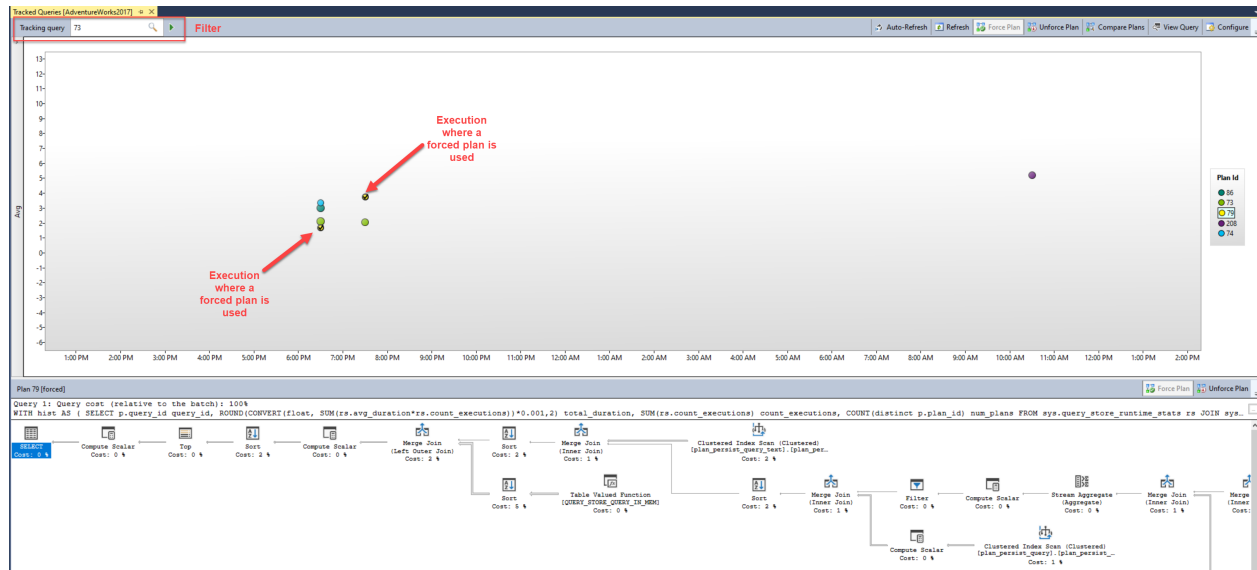
The **Query Wait Statistics** view analyzes the most active wait categories for the database and renders a chart. This chart is interactive; selecting a wait category drills into the details of the queries that contribute to the wait time statistic.



The details view interface is also consistent with most query store views allowing for query detail inspection, execution plan evaluation, and optionally forcing a specific plan. This view helps identify queries that are affecting user experience across applications.

Tracking Query

The **Tracking Query** view allows analyzing a specific query based on an entered query ID value. Once run, the view provides the complete execution history of the query. A checkmark on an execution indicates a forced plan was used. This view can provide insight into queries such as those with forced plans to verify that query performance is remaining stable.

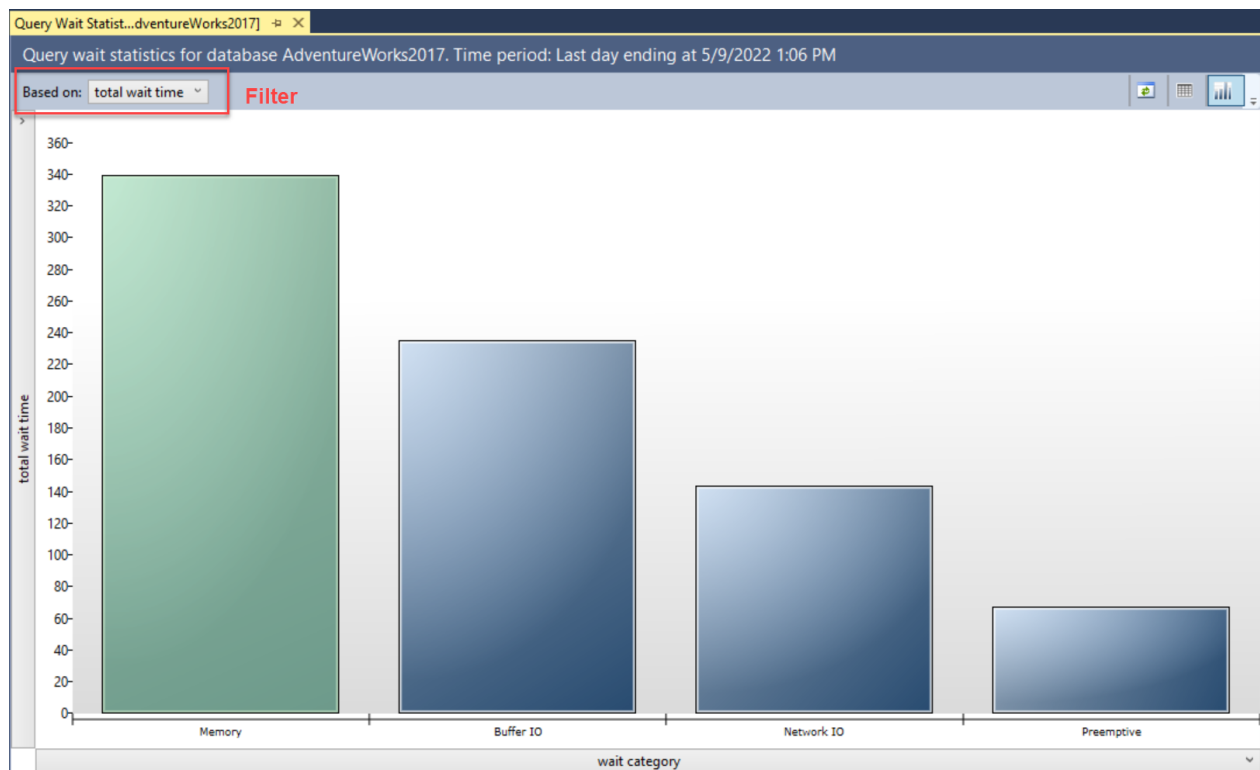


Using the Query Store to find query waits

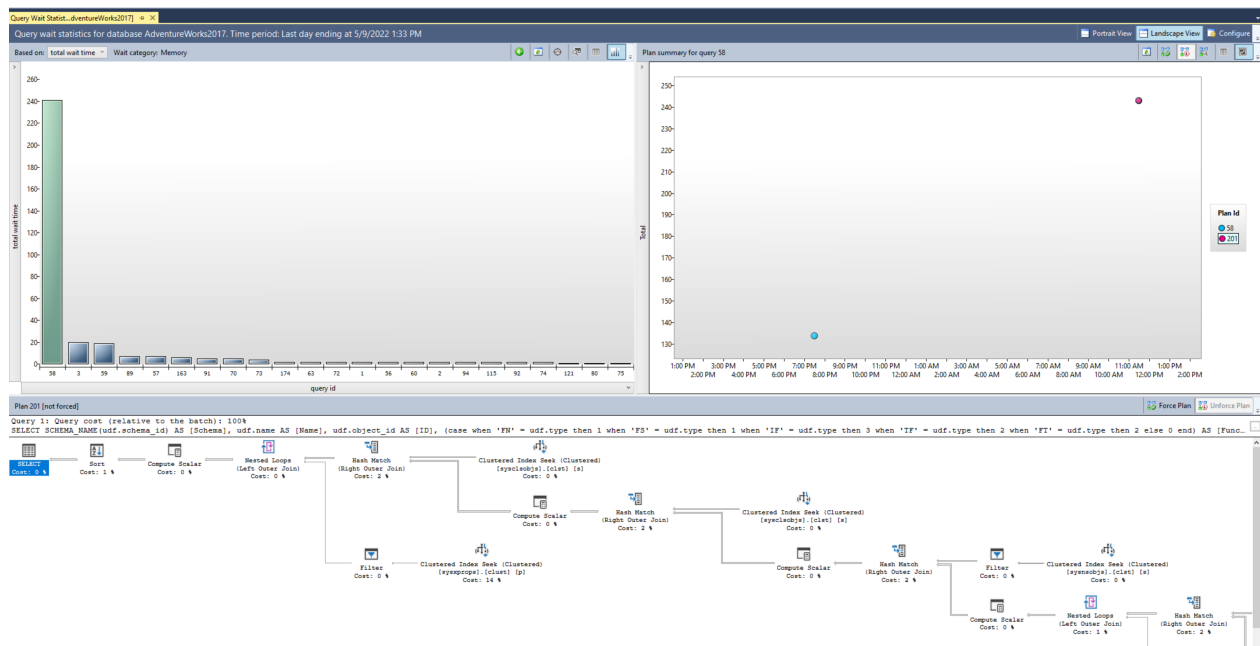
When the performance of a system begins to degrade, it makes sense to consult query wait statistics to potentially identify a cause. In addition to identifying queries that need to tune, it can also shed light on potential infrastructure upgrades that would be beneficial.

The SQL Query Store provides the **Query Wait Statistics** view to provide insight into the top wait categories for the database. Currently, there are [23 wait categories](#).

A bar chart displays the most impactful wait categories for the database when you open the Query Wait Statistics view. In addition, a filter located in the toolbar of the wait categories pane allows for the wait statistics to be calculated based on total wait time (default), average wait time, minimum wait time, maximum wait time, or standard deviation wait time.



Selecting a wait category drills through to the details of the queries that contribute to that wait category. From this view, you have the ability to investigate individual queries that are the most impactful. You can access the persisted estimated execution plans display in the Plan summary pane by selecting a query in the query pane. Selecting a query plan from the Plan summary pane displays the graphical query plan in the bottom pane. From this view, you have the ability to force or unforce a query plan for the query to improve performance.



Automatic plan correction

SQL Server 2017 and Azure SQL Database introduced the concept of automatic plan correction by analyzing data in the Query Store. When you enable the Query Store with a database in SQL Server 2017 (or later) and in Azure SQL Database, the SQL Server engine looks for query plan regressions and provides recommendations. You can see these recommendations in the `sys.dm_db_tuning_recommendations` dynamic management view (DMV). These recommendations include T-SQL statements to manually force a query plan when performance was in a good state.

If you gain confidence in these recommendations, you can enable SQL Server to force plans automatically when regressions are encountered. Enable automatic plan correction by using `ALTER DATABASE` and the `AUTOMATIC_TUNING` argument.

For Azure SQL Database, you can also enable automatic plan correction through automatic tuning options in the Azure portal or REST APIs. Automatic plan correction recommendations are always enabled for any database where Query Store is enabled (which is the default for Azure SQL Database and Azure SQL Managed Instance). For new databases, automatic plan correction (`FORCE_PLAN`) is enabled by default for Azure SQL Database.

6. Identify problematic query plans

<https://learn.microsoft.com/en-us/training/modules/explore-query-performance-optimization/6-identify-problematic>

Identify problematic query plans

Completed

The typical approach DBAs take to troubleshoot query performance involves first identifying the problematic query, usually the one consuming the most system resources, and then retrieving its execution plan. There are two main scenarios. One scenario is that the query consistently performs poorly. This can be due to various issues, such as hardware resource constraints (though this usually doesn't affect a single query running in isolation), suboptimal query structure, database compatibility settings, missing indexes, or poor plan choices by the query optimizer. The second scenario is that the query performs well in some executions but poorly in others. This inconsistency can be caused by factors like data skew in a parameterized query, which has an efficient plan for some executions and a poor one for others. Other common factors include blocking, where a query waits for another query to complete to gain access to a table, or hardware contention.

Let's explore each of these scenarios in more detail.

Hardware constraints

Hardware constraints typically don't manifest during single query executions but become evident under production load when CPU threads and memory are limited. CPU contention can be detected by observing the performance monitor counter '% Processor Time', which measures server CPU usage. In SQL Server, `SOS_SCHEDULER_YIELD` and `CXPACKET` wait types can indicate CPU pressure. Poor storage system performance can slow down even optimized single query executions. Storage performance is best tracked at the operating system level using performance monitor counters `Disk Seconds/Read` and `Disk Seconds/Write`, which measure I/O operation completion times. SQL Server logs poor storage performance if an I/O takes longer than 15 seconds. High `PAGEIOLATCH_SH` waits in SQL Server can indicate storage performance issues. Hardware performance is typically evaluated early in the troubleshooting process due to its ease of assessment.

Most database performance issues stem from suboptimal query patterns, which can put undue pressure on hardware. For example, missing indexes can lead to CPU, storage, and memory pressure by retrieving more data than necessary. It's recommended to address and tune suboptimal queries before tackling hardware issues. Next, we look at query tuning.

Suboptimal query constructs

Relational databases perform best when executing set-based operations, which manipulate data (`INSERT` , `UPDATE` , `DELETE` , and `SELECT`) in sets, producing either a single value or a result set. The alternative is row-based processing, using cursors or while loops, which increase cost linearly with the number of rows impacted—a problematic scale as data volumes grow.

Detecting suboptimal use of row-based operations with cursors or `WHILE` loops is important, but there are other SQL Server anti-patterns to recognize. Table-valued functions (TVFs), particularly multi-statement TVFs, caused problematic execution plan patterns before SQL Server 2017. Developers often use multi-statement TVFs to execute multiple queries within a single function and aggregate results into a single table. However, using TVFs can lead to performance penalties.

SQL Server has two types of TVFs: inline and multi-statement. Inline TVFs are treated like views, while multi-statement TVFs are treated like tables during query processing. Because TVFs are dynamic and lack statistics, SQL Server uses a fixed row count for estimating query plan cost. This can be fine for small row counts, but inefficient for thousands or millions of rows.

Another anti-pattern is the use of scalar functions, which have similar estimation and execution problems. Microsoft has made significant performance improvements with Intelligent Query Processing, under compatibility levels 140 and 150.

SARGability

The term SARGable in relational databases refers to a predicate (`WHERE` clause) formatted to use an index to speed up query execution. Predicates in the correct format are called 'Search Arguments' or SARGs. In SQL Server, using a SARG means the optimizer evaluates using a nonclustered index on the column referenced in the SARG for a `SEEK` operation, instead of scanning the entire index or table to retrieve a value.

The presence of a SARG doesn't guarantee the use of an index for a *SEEK*. The optimizer's costing algorithms could still determine that the index is too expensive, especially if a SARG refers to a large percentage of rows in a table. The absence of a SARG means the optimizer won't evaluate a *SEEK* on a nonclustered index.

Examples of non-SARGable expressions include those with a `LIKE` clause using a wildcard at the beginning of the string, such as `WHERE LastName LIKE '%SMITH%'`. Other non-SARGable predicates occur when using functions on a column, like `WHERE CONVERT(CHAR(10), CreateDate, 121) = '2020-03-22'`. These queries are typically identified by examining execution plans for index or table scans where seeks should otherwise occur.

SQLQuery2.sql - VM...VM1\jdantoni (88)* ExecutionPlan1.sqlplan*

```

SELECT AddressID, AddressLine1, AddressLine2, City
FROM Person.Address
WHERE LEFT (City,1) = 'M'

```

100 %

Messages Execution plan

Query 1: Query cost (relative to the batch): 100%
 SELECT AddressID, AddressLine1, AddressLine2, City FROM Person.Address WHERE LEFT (City,1) = 'M'

SELECT
 Cost: 1 %

Index Scan (NonClustered)
 [Address].[IX_Address_City]
 Cost: 99 %

Index Scan (NonClustered)	
Scan a nonclustered index, entirely or only a range.	
Physical Operation	Index Scan
Logical Operation	Index Scan
Estimated Execution Mode	Row
Storage	RowStore
Estimated Operator Cost	0.148561 (99%)
Estimated I/O Cost	0.126829
Estimated Subtree Cost	0.148561
Estimated CPU Cost	0.0217324
Estimated Number of Executions	1
Estimated Number of Rows	1216
Estimated Number of Rows to be Read	19614
Estimated Row Size	156 B
Ordered	False
Node ID	1
Predicate substring([AdventureWorks2017].[Person].[Address].[City],(1),(1))='M'	
Object [AdventureWorks2017].[Person].[Address].[IX_Address_City]	
Output List [AdventureWorks2017].[Person].[Address].AddressID, [AdventureWorks2017].[Person].[Address].AddressLine1, [AdventureWorks2017].[Person].[Address].AddressLine2, [AdventureWorks2017].[Person].[Address].City	

There's an index on the `City` column that is being used in the `WHERE` clause of the query and while it's being used in this execution plan above, you can see the index is being scanned, which means the

entire index is being read. The `LEFT` function in the predicate makes this expression non-SARGable. The optimizer won't evaluate using an index seek on the index on the `City` column.

This query could be written to use a predicate that is SARGable. The optimizer would then evaluate a *SEEK* on the index on the `City` column. An index seek operator, in this case, would read a smaller set of rows.

SQLQuery2.sql - VM...VM1\jdantoni (88)* -> X ExecutionPlan1.sqlplan*

```
SELECT AddressID, AddressLine1, AddressLine2, City
FROM Person.Address
WHERE City LIKE 'M%'
```

100 %

Messages Execution plan

Query 1: Query cost (relative to the batch): 100%

SELECT AddressID, AddressLine1, AddressLine2, City FROM Person.Address WHERE City LIKE 'M%'

Index Seek (NonClustered)
[Address].[IX_Address_City]
Cost: 100 %

SELECT
Cost: 0 %

Index Seek (NonClustered)	
Scan a particular range of rows from a nonclustered index.	
Physical Operation	Index Seek
Logical Operation	Index Seek
Estimated Execution Mode	Row
Storage	RowStore
Estimated Operator Cost	0.0120842 (100%)
Estimated I/O Cost	0.0105324
Estimated Subtree Cost	0.0120842
Estimated CPU Cost	0.0015518
Estimated Number of Executions	1
Estimated Number of Rows	1267.98
Estimated Number of Rows to be Read	1267.98
Estimated Row Size	169 B
Ordered	True
Node ID	0
Predicate [AdventureWorks2017].[Person].[Address].[City] like N'M%'	
Object [AdventureWorks2017].[Person].[Address].[IX_Address_City]	
Output List [AdventureWorks2017].[Person].[Address].AddressID, [AdventureWorks2017].[Person].[Address].AddressLine1, [AdventureWorks2017].[Person].[Address].AddressLine2, [AdventureWorks2017].[Person].[Address].City	
Seek Predicates Seek Keys[1]: Start: [AdventureWorks2017].[Person]. [Address].City >= Scalar Operator(N'M'), End: [AdventureWorks2017].[Person].[Address].City < Scalar Operator(N'N')	

Changing `LEFT` function into a `LIKE` results in an index seek.

Note

The `LIKE` keyword, in this example, doesn't have a wildcard on the left, so it's looking for cities that begin with M. If it was "two-sided" or started with a wildcard ('%M%' or '%M') it would be non-SARGable. The seek operation is estimated to return 1,267 rows, or approximately 15% of the estimate for the query with the non-SARGable predicate.

Some other database development anti-patterns are treating the database as a service rather than a data store. Using a database to convert data to JSON, manipulate strings, or perform complex calculations can lead to excessive CPU use and increased latency. Queries that try to retrieve all records and then perform computations in the database can lead to excessive IO and CPU usage. Ideally, you should use the database for data access operations and optimized database constructs like aggregation.

Missing indexes

The most common performance problems for database administrators stem from a lack of useful indexes, causing the engine to read more pages than necessary to return query results. While indexes consume resources (affecting write performance and consuming space), their performance gains often outweigh the extra resource costs. Execution plans with these issues can be identified by the query operator *Clustered Index Scan* or the combination of *Nonclustered Index Seek* and *Key Lookup*, indicating missing columns in an existing index.

The database engine helps by reporting missing indexes in execution plans. The names and details of recommended indexes are available through the dynamic management view

`sys.dm_db_missing_index_details`. Other DMVs like `sys.dm_db_index_usage_stats` and `sys.dm_db_index_operational_stats` highlight the utilization of existing indexes.

Dropping an unused index can be sensible. Missing index DMVs and plan warnings should be starting points for tuning queries. It's crucial to understand key queries and build indexes to support them. Creating all missing indexes without evaluating them in context isn't recommended.

Missing and out-of-date statistics

Understanding the importance of column and index statistics to the query optimizer is crucial. It's also essential to recognize conditions that can lead to out-of-date statistics and how this issue can manifest in SQL Server. Azure SQL offerings default to having autoupdate statistics set to ON. Before SQL Server 2016, the default behavior of autoupdate statistics was to not update statistics until the number of modifications to columns in the index equaled about 20% of the number of rows in a table. This behavior could result in significant data modifications that change query performance without updating the statistics, leading to suboptimal plans based on outdated statistics.

Before SQL Server 2016, trace flag 2371 could be used to change the required number of modifications to a dynamic value, so as your table grew, the percentage of row modifications needed

to trigger a statistics update decreased. Newer versions of SQL Server, Azure SQL Database, and Azure SQL Managed Instance support this behavior by default. The dynamic management function `sys.dm_db_stats_properties` shows the last time statistics were updated and the number of modifications since the last update, allowing you to quickly identify statistics that might need manual updates.

Poor optimizer choices

While the query optimizer does a good job of optimizing most queries, there are some edge cases where the cost-based optimizer can make impactful decisions that aren't fully understood. There are many ways to address this including using query hints, trace flags, execution plan forcing, and other adjustments in order to reach a stable and optimal query plan. Microsoft has a support team that can help troubleshoot these scenarios.

In the below example from the *AdventureWorks2017* database, a query hint is being use to tell the database optimizer to always use a city name of Seattle. This hint won't guarantee the best execution plan for all city values, but it's predictable. The value of 'Seattle' for `@city_name` will only be used during optimization. During execution, the actual supplied value ('Ascheim') is used.

```
DECLARE @city_name nvarchar(30) = 'Ascheim',
        @postal_code nvarchar(15) = 86171;

SELECT *
FROM Person.Address
WHERE City = @city_name
      AND PostalCode = @postal_code
OPTION (OPTIMIZE FOR (@city_name = 'Seattle'));
```

As seen in the example, the query uses a hint (the `OPTION` clause) to tell the optimizer to use a specific variable value to build its execution plan.

Parameter sniffing

SQL Server caches query execution plans for future use. Since the execution plan retrieval process is based on the hash value of a query, the query text has to be identical for every execution of the query for the cached plan to be used. In order to support multiple values in the same query, many developers use parameters, passed in through stored procedures, as seen in the following example:

```
CREATE PROC GetAccountID (@Param INT)
AS

<other statements in procedure>

SELECT accountid FROM CustomerSales WHERE sales > @Param;

<other statements in procedure>
```

```

RETURN;

-- Call the procedure:

EXEC GetAccountID 42;

```

Queries can also be explicitly parameterized using the procedure `sp_executesql`. However, explicit parameterization of individual queries is done through the application with some form (depending on the API) of *PREPARE* and *EXECUTE*. When the database engine executes that query for the first time, it optimizes the query based on the initial value of the parameter, in this case, 42. This behavior, called parameter sniffing, allows for the overall workload of compiling queries to be reduced on the server. However, if there's data skew, query performance could vary widely.

For example, a table that had 10 million records, and 99% of those records have an ID of 1, and the other 1% are unique numbers, performance is based on which ID was initially used to optimize the query. This wildly fluctuating performance is indicative of data skew and isn't an inherent problem with parameter sniffing. This behavior is a fairly common performance problem that you should be aware of. You should understand the options for alleviating the issue. There are a few ways to address this problem, but they each come with tradeoffs:

- Use the `RECOMPILE` hint in your query, or the `WITH RECOMPILE` execution option in your stored procedures. This hint causes the query or procedure to be recompiled every time it's executed, which will increase CPU utilization on the server but will always use the current parameter value.
- You can use the `OPTIMIZE FOR UNKNOWN` query hint. This hint causes the optimizer to choose to not sniff parameters and compare the value with column data histogram. This option won't get you the best possible plan but will allow for a consistent execution plan.
- Rewrite your procedure or queries by adding logic around parameter values to only `RECOMPILE` for known troublesome parameters. In the example below, if the `SalesPersonID` parameter is NULL, the query is executed with the `OPTION (RECOMPILE)`.

```

CREATE OR ALTER PROCEDURE GetSalesInfo (@SalesPersonID INT = NULL)
AS
DECLARE  @Recompile BIT = 0
         , @SQLString NVARCHAR(500)

SELECT @SQLString = N'SELECT SalesOrderId, OrderDate FROM Sales.SalesOrderHeader W

IF @SalesPersonID IS NULL
BEGIN
    SET @Recompile = 1
END

IF @Recompile = 1
BEGIN
    SET @SQLString = @SQLString + N' OPTION(RECOMPILE)'
END

EXEC sp_executesql @SQLString
    ,N'@SalesPersonID INT'

```

```
,@SalesPersonID = @SalesPersonID  
GO
```

This example is a good solution but it does require a fairly large development effort, and a firm understanding of your data distribution. It requires maintenance as the data changes.

7. Describe blocking and locking

<https://learn.microsoft.com/en-us/training/modules/explore-query-performance-optimization/7-describe-blocking-locking>

Describe blocking and locking

Completed

Locking is a key feature of relational databases, essential for maintaining the atomicity, consistency, and isolation properties of the ACID model. All RDBMSs block actions that would violate the consistency and isolation of database writes. SQL programmers must start and end transactions at the right points to ensure data consistency. The database engine provides locking mechanisms to protect the logical consistency of the affected tables, which is foundational to the relational model.

In SQL Server, blocking occurs when one process holds a lock on a specific resource (row, page, table, database), and a second process attempts to acquire a lock with an incompatible lock type on the same resource. Typically, locks are held for a short period, and once the process holding the lock releases it, the blocked process can acquire the lock and complete its transaction.

SQL Server locks the smallest amount of data needed to complete a transaction, allowing for maximum concurrency. For example, if SQL Server locks a single row, all other rows in the table remain available for other processes, enabling concurrent work. However, each lock requires memory resources, so it's not cost-effective for one process to hold thousands of individual locks on a single table. To balance concurrency with cost, SQL Server uses a technique called lock escalation. If more than 5,000 rows on a single object need to be locked in a single statement, SQL Server escalates the multiple row locks to a single table lock.

Locking is a normal behavior and occurs frequently throughout the day. It only becomes problematic when it causes blocking that isn't quickly resolved. There are two types of performance issues caused by blocking:

- A process holds locks on a set of resources for an extended period before releasing them, causing other processes to block and degrading query performance and concurrency.
- A process acquires locks on a set of resources and never releases them, requiring administrator intervention to resolve.

Deadlocking is another blocking scenario that occurs when one transaction holds a lock on a resource, and another transaction holds a lock on a different resource. Each transaction then attempts to acquire a lock on the resource currently locked by the other transaction, leading to an infinite wait as neither transaction can complete. The SQL Server engine detects these scenarios and resolves the deadlock by killing one of the transactions, based on which transaction has performed the least amount of work that needs to be rolled back. The transaction that is killed is known as the deadlock victim. Deadlocks are recorded in the *system_health* extended event session, which is enabled by default.

It's important to understand the concept of a transaction. Autocommit is the default mode of SQL Server and Azure SQL Database, which means the changes made by the following statement would automatically be recorded in the database's transaction log.

```
INSERT INTO DemoTable (A) VALUES (1);
```

In order to allow developers to have more granular control over their application code, SQL Server also allows you to explicitly control your transactions. The following query would take a lock on a row in the *DemoTable* table that wouldn't be released until a subsequent command to commit the transaction was added.

```
BEGIN TRANSACTION  
  
INSERT INTO DemoTable (A) VALUES (1);
```

The proper way to write the following query is as follows:

```
BEGIN TRANSACTION  
  
INSERT INTO DemoTable (A) VALUES (1);  
  
COMMIT TRANSACTION
```

The `COMMIT TRANSACTION` command explicitly commits a record of the changes to the transaction log. The changed data will eventually make its way into the data file asynchronously. These transactions represent a unit of work to the database engine. If the developer forgets to issue the `COMMIT TRANSACTION` command, the transaction stays open and the locks won't be released. This is one of the main reasons for long running transactions.

The other mechanism the database engine uses to help the concurrency of the database is row versioning. When a row versioning isolation level is enabled to the database, the engine maintains versions of each modified row in TempDB. This is typically used in mixed use workloads, in order to prevent reading queries from blocking queries that are writing to the database.

To monitor open transactions awaiting commit or rollback run the following query:

```
SELECT tst.session_id, [database_name] = db_name(s.database_id)  
      , tat.transaction_begin_time
```

```

, transaction_duration_s = datediff(s, tat.transaction_begin_time, sysdatetime())
, transaction_type = CASE tat.transaction_type WHEN 1 THEN 'Read/write transaction'
  WHEN 2 THEN 'Read-only transaction'
  WHEN 3 THEN 'System transaction'
  WHEN 4 THEN 'Distributed transaction' END
, input_buffer = ib.event_info, tat.transaction_uow
, transaction_state = CASE tat.transaction_state
  WHEN 0 THEN 'The transaction has not been completely initialized yet.'
  WHEN 1 THEN 'The transaction has been initialized but has not started.'
  WHEN 2 THEN 'The transaction is active - has not been committed or rolled back.'
  WHEN 3 THEN 'The transaction has ended. This is used for read-only transactions.'
  WHEN 4 THEN 'The commit process has been initiated on the distributed transaction.'
  WHEN 5 THEN 'The transaction is in a prepared state and waiting for resolution.'
  WHEN 6 THEN 'The transaction has been committed.'
  WHEN 7 THEN 'The transaction is being rolled back.'
  WHEN 8 THEN 'The transaction has been rolled back.' END
, transaction_name = tat.name, request_status = r.status
, tst.is_user_transaction, tst.is_local
, session_open_transaction_count = tst.open_transaction_count
, s.host_name, s.program_name, s.client_interface_name, s.login_name, s.is_user
FROM sys.dm_tran_active_transactions tat
INNER JOIN sys.dm_tran_session_transactions tst ON tat.transaction_id = tst.transaction_id
INNER JOIN sys.dm_exec_sessions s ON s.session_id = tst.session_id
LEFT OUTER JOIN sys.dm_exec_requests r ON r.session_id = s.session_id
CROSS APPLY sys.dm_exec_input_buffer(s.session_id, null) AS ib
ORDER BY tat.transaction_begin_time DESC;

```

Isolation levels

SQL Server offers several isolation levels to allow you to define the level of consistency and correctness you need guaranteed for your data. Isolation levels let you find a balance between concurrency and consistency. The isolation level doesn't affect the locks taken to prevent data modification. A transaction will always get an exclusive lock on the data that is modifying. However, your isolation level can affect the length of time that your locks are held. Lower isolation levels increase the ability of multiple user process to access data at the same time, but increase the data consistency risks that can occur. The isolation levels in SQL Server are as follows:

- **Read uncommitted** – Lowest isolation level available. Dirty reads are allowed, which means one transaction may see changes made by another transaction that haven't yet been committed.
- **Read committed** – allows a transaction to read data previously read, but not modified by another transaction without waiting for the first transaction to finish. This level also releases read locks as soon as the select operation is performed. This is the default SQL Server level.
- **Repeatable Read** – This level keeps read and write locks that are acquired on selected data until the end of the transaction.

- **Serializable** – This is the highest level of isolation where transactions are isolated. Read and write locks are acquired on selected data and not released until the end of the transaction.

SQL Server also includes two isolation levels that include row-versioning.

- **Read Committed Snapshot** – In this level read operations take no row or page locks, and the engine presents each operation with a consistent snapshot of the data as it existed at the start of the query. This level is typically used when users are running frequent reporting queries against an OLTP database, in order to prevent the read operations from blocking the write operations.
- **Snapshot** – This level provides transaction level read consistency through row versioning. This level is vulnerable to update conflicts. If a transaction running under this level reads data modified by another transaction, an update by the snapshot transaction will be terminated and roll back. This isn't an issue with read committed snapshot isolation.

Isolation levels are set for each session with the T-SQL `SET` command, as shown:

```
SET TRANSACTION ISOLATION LEVEL  
  
{ READ UNCOMMITTED  
  
  | READ COMMITTED  
  
  | REPEATABLE READ  
  
  | SNAPSHOT  
  
  | SERIALIZABLE  
  
}
```

There's no way to set a global isolation level all queries running in a database, or for all queries run by a particular user. It's a session level setting.

Monitoring for blocking problems

Identifying blocking problems can be challenging due to their sporadic nature. The DMV `sys.dm_tran_locks`, when joined with `sys.dm_exec_requests`, provides information on the locks held by each session. A more effective way to monitor blocking problems is to use the Extended Events engine on an ongoing basis.

Blocking problems typically fall into two categories:

- Poor transactional design: For example, a transaction without a `COMMIT TRANSACTION` will never end. Trying to do too much work in a single transaction or having a distributed transaction using a linked server connection can lead to unpredictable performance.

- Long-running transactions caused by schema design: This often involves an update on a column with a missing index or a poorly designed update query.

Monitoring for locking-related performance problems allows you to quickly identify performance degradation related to locking.

For more information about how to monitor blocking, see [Understand and resolve SQL Server blocking problems](#).

8. Exercise: Identify and resolve blocking issues

<https://learn.microsoft.com/en-us/training/modules/explore-query-performance-optimization/8-exercise-identify-resolve-blocking-issues>

Exercise: Identify and resolve blocking issues

Completed

In this exercise, you'll learn how to identify and troubleshoot blocking issues using SQL Server Management Studio (SSMS).

Note

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

Launch the exercise and follow the instructions.

[Launch Exercise](#)

9. Module assessment

<https://learn.microsoft.com/en-us/training/modules/explore-query-performance-optimization/9-knowledge-check>

Module assessment

Completed

10. Summary

<https://learn.microsoft.com/en-us/training/modules/explore-query-performance-optimization/10-summary>

Summary

Completed

Database performance is determined by how execution plans work and how indexes are used. At the same time, you need to understand patterns and anti-patterns for relational databases and aim for good query choices and proper indexing. Query Store acts as a data recorder for the database engine, and it is focused on query performance and changes in execution plans.

Additional reading

- [Azure SQL Database overview](#)
- [Live Query Statistics](#)
- [Best practices for monitoring workloads with Query Store](#)
- [How Query Store collects data](#)