**Understand query plans**

It's helpful to have a basic understanding of how database optimizers work before taking a deeper dive into execution plan details. SQL Server uses what is known as cost-based query optimizer. The query optimizer calculates a cost for multiple possible plans based on the statistics it has on the columns being utilized, and the possible indexes that can be used for each operation in each query plan. Based on this information, it comes up with a total cost for each plan. Some complex queries can have thousands of possible execution plans. The optimizer doesn't evaluate every possible plan, but uses heuristics to determine plans that are likely to have good performance. The optimizer will then choose the lowest cost plan of all the plans evaluated for a given query.

Because the query optimizer is cost-based, it's important that it has good inputs for decision making. The statistics SQL Server uses to track the distribution of data in columns and indexes need be kept up to date, or it can cause suboptimal execution plans to be generated. SQL Server automatically updates its statistics as data changes in a table; however, more frequent updates may be needed for rapidly changing data. The engine uses many factors when building a plan including compatibility level of the database, 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 a parse tree of database objects is generated if the syntax is correct.
2. The parse tree from Step 1 is taken as 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 that are being processed for a given query. This step outputs a query processor tree, which is in the input for step 3.
3. Because query optimization is a relatively expensive process in terms of CPU consumption, the database engine caches execution plans in a special area of memory called the plan cache. If a plan for a given query already exists, that plan is retrieved from the cache. The queries whose plans are stored in cache will each have a hash value generated based on the T-SQL in the query. This value is referred to as the query\_hash. The engine will generate a *query\_hash* for the current query and then look to see if it matches any existing queries in the plan cache.
4. If the plan doesn't exist, the Query Optimizer then uses its cost-based optimizer to generate several execution plan options based on the statistics about the columns, tables, and indexes that are used in the query, as described above. The output of this step is a query execution plan.
5. The query is then executed using an execution plan that is pulled from the plan cache, or a new plan generated in step 4. The output of this step is the results of your query.

**Note**

To learn more about how the query processor works, see [**Query Processing Architecture Guide**](https://learn.microsoft.com/en-us/sql/relational-databases/query-processing-architecture-guide)

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

SQLCopy

SELECT orderdate,

AVG(salesAmount)

FROM FactResellerSales

WHERE ShipDate = '2013-07-07'

GROUP BY orderdate;

In this example SQL Server will check for the existence of the *OrderDate*, *ShipDate*, and *SalesAmount* columns in the table *FactResellerSales*. If those columns exist, it will then generate a hash value for the query, and examine the plan cache for a matching hash value. If there's plan for a query with a matching hash the engine will try to reuse that plan. If there's no plan with a matching hash, it will examine the statistics it has available on the *OrderDate* and *ShipDate* columns. The WHERE clause referencing the *ShipDate* column is what is known as the predicate in this query. If there's a nonclustered index that includes the *ShipDate* column SQL Server will most likely include that in the plan, if the costs are lower than retrieving data from the clustered index. The optimizer will then choose the lowest cost plan of the available plans and execute the query.

Query plans combine a series of relational operators to retrieve the data, and also capture information about the data such as estimated row counts. Another element of the execution plan is the memory required to perform operations such as joining or sorting data. The memory needed by the query is called the memory grant. The memory grant is a good example of the importance of statistics. If SQL Server thinks an operator is going to return 10,000,000 rows, when it's only returning 100, a much larger amount of memory is granted to the query. A memory grant that is larger than necessary can cause a twofold problem. First, the query may encounter a RESOURCE\_SEMAPHORE wait, which indicates that query is waiting for SQL Server to allocate it 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, when the query is executed, if there isn't enough memory available, the query will spill to tempdb, which is much slower than operating in memory.

The execution plan also stores other metadata about the query, including, but not limited to, the database compatibility level, the degree of parallelism of the query, and the parameters that are supplied if the query is parameterized.

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

Most DBAs prefer to look at plans graphically, because a graphical plan allows you to see the plan as a whole, including what’s called the *shape* of the plan, easily. There are several ways you can view and save graphical query plans. The most common tool used for this purpose is SQL Server Management Studio, but estimated plans can also be viewed in Azure Data Studio. There are also third-party tools that support viewing graphical execution plans.

There are three different types of execution plans that can be viewed.

**Estimated Execution Plan**

This type is the execution plan as generated by the query optimizer. The metadata and size of query memory grant are based on estimates from the statistics as they exist in the database at the time of query compilation. To see a text-based estimated plan run the command SET SHOWPLAN\_ALL ON before running the query. When you run the query, you'll see the steps of the execution plan, but the query will NOT be executed, and you won't see any results. The SET option will stay in effect until you set it OFF.

**Actual Execution Plan**

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

**Live Query Statistics**

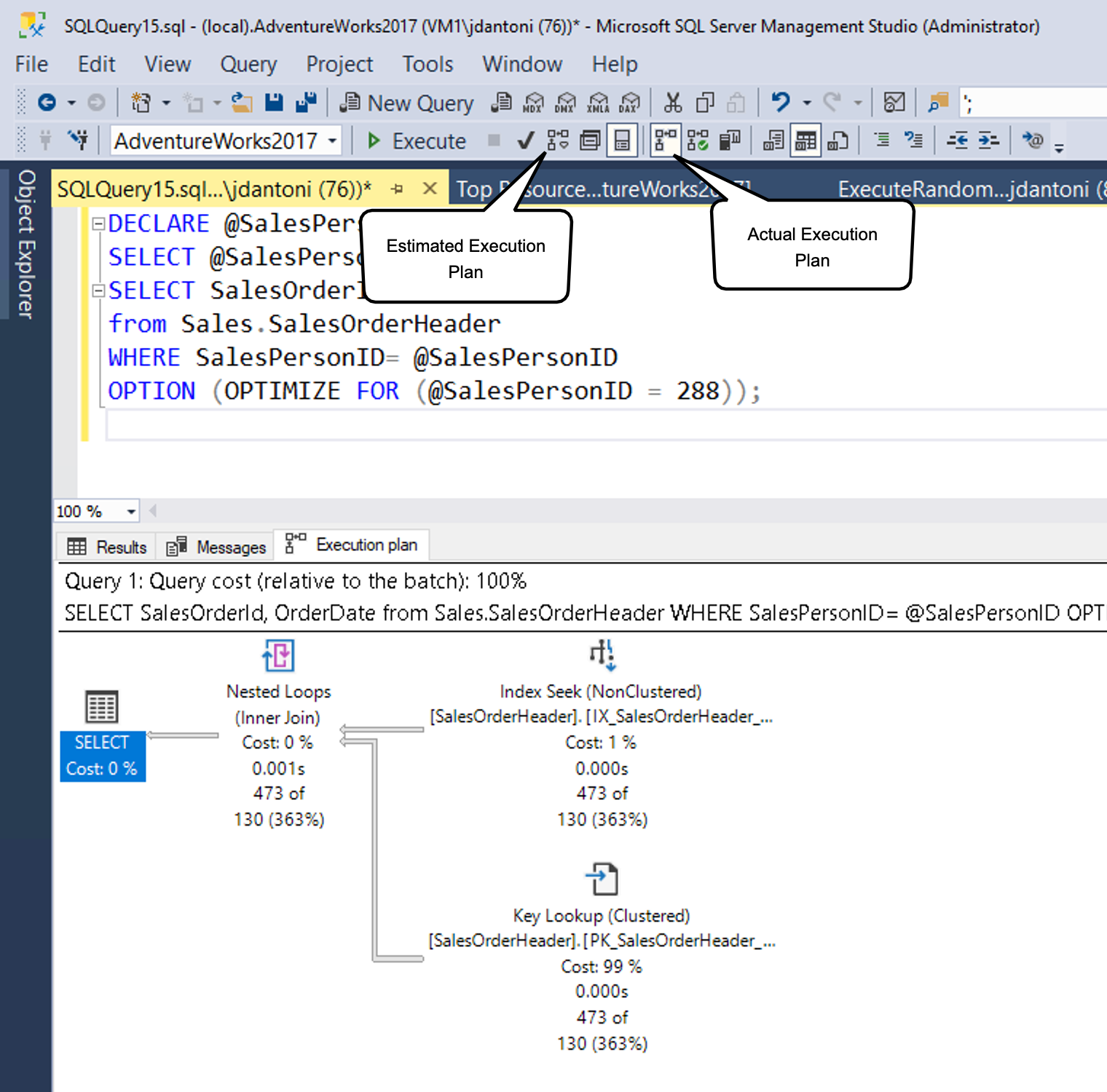
This plan viewing option combines the estimated and actual plans into an animated plan that displays execution progress through the operators in the plan. It refreshes every second and shows the actual number of rows flowing through the operators. The other benefit to Live Query Statistics is that it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

# Explain estimated and actual query plans

Completed100 XP

* 13 minutes

The topic of 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 order of execution will be the same as the estimated plan in nearly all cases. The other consideration is that in order to capture an actual execution plan the query has to be executed, which can be time consuming, or not possible. For example, the query may be an UPDATE statement that can only be run once. However, if you need to see query results and the plan, you’ll need to use one of the actual plan options.



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

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.

Usually 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.

First, the query itself is shown below:

Transact-SQLCopy

SELECT [stockItemName]

,[UnitPrice] \* [QuantityPerOuter] AS CostPerOuterBox

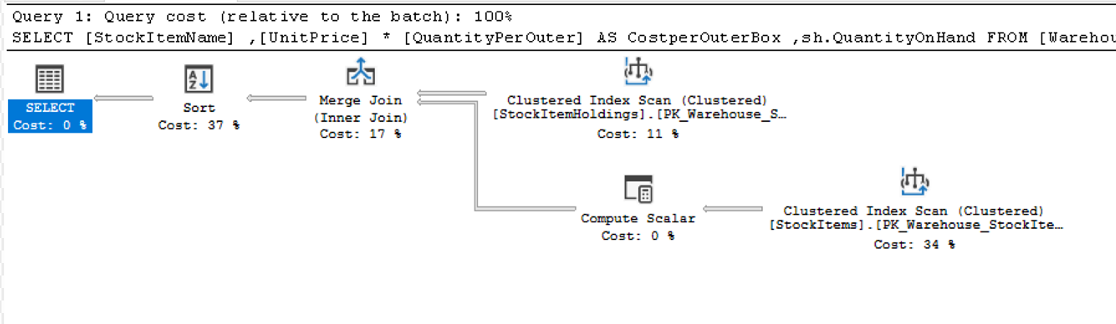
,[QuantityonHand]

FROM [Warehouse].[StockItems] s

JOIN [Warehouse].[StockItemHoldings] sh ON s.StockItemID = sh.StockItemID

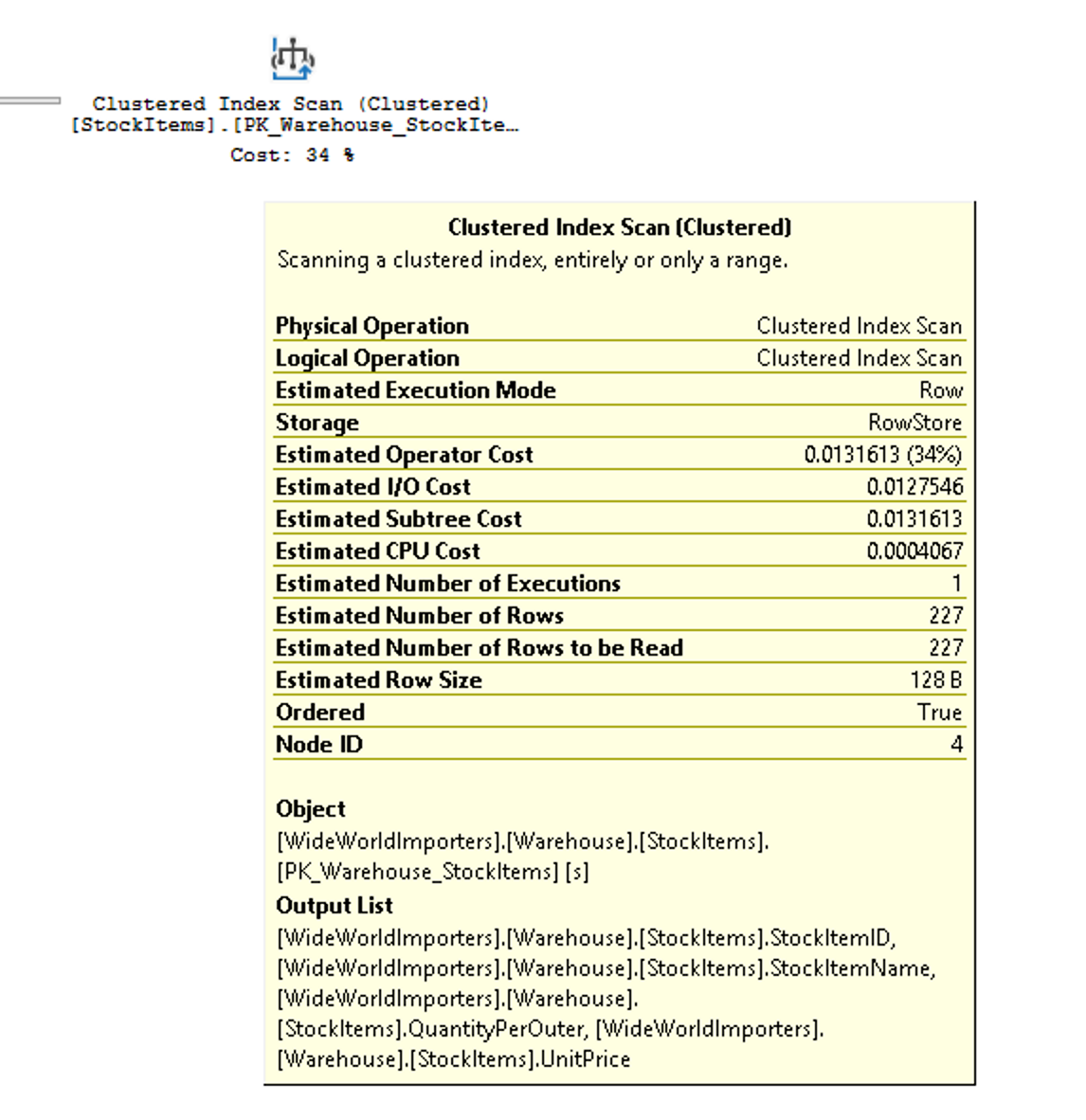
ORDER BY CostPerOuterBox;

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.

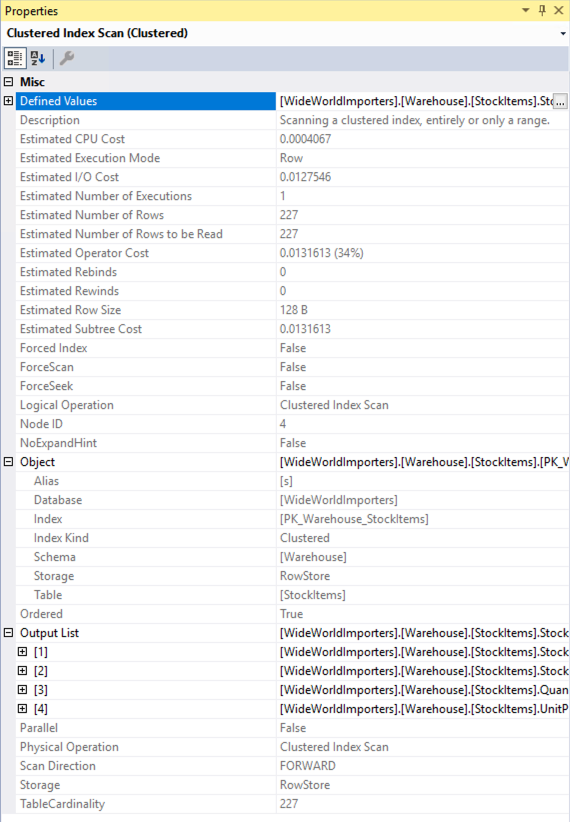


Each icon in the plan shows a specific operation, which represents the various actions and decisions that make up an execution plan. The SQL Server database engine has over 100 query operators that can make up on an execution plan. You'll notice that under each operator icon, there's a cost percentage relative to the total cost of the query. Even an operation that shows a cost of 0% still represents some cost. In fact, 0% is usually due to rounding, because the 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 the plan above, 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 as shown below.



The tooltip highlights the cost and estimates for the estimated plan, and for an actual plan will include the comparisons to the actual rows and costs. Each operator also has properties that will show you more than the tooltip does. If you right-click on a specific operator, you can select the Properties option from the context menu to see the full property list. This option will open up a separate Properties pane in SQL Server Management Studio, which by default is on the right side. Once the Properties pane is open, clicking on any operator will populate the Properties list with properties for that operator. Alternatively, you can open the Properties pane by clicking on View in the main SQL Server Management Studio menu and choosing Properties.



The Properties pane includes some additional information and shows the output list, which provides details of 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 clustered index scan. Since a clustered index scan operation is reading the entire table, in this scenario a non-clustered index on the StockItemID column in each table could be more efficient.

## Lightweight query profiling

As mentioned above, capturing actual execution plans, whether using SSMS or the Extended Events monitoring infrastructure can have a large amount of overhead, and is typically only done in 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 slow down individual query performance significantly. The legacy profiling infrastructure in SQL Server’s engine could produce up to 75% overhead for capturing query information, whereas the lightweight profiling infrastructure 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). In addition, 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.

In newer releases (SQL Server 2016 SP2 CU3, SQL Server 2017 CU11, and SQL Server 2019), 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 that has this hint completes execution, a query\_plan\_profile extended event is generated, which provides an actual execution plan. You can see an example of a query with this hint:

Transact-SQLCopy

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

SQL Server 2019 and Azure SQL Database support two further enhancements to the query profiling infrastructure. First, 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.

Second, there's a new 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 as shown below:

Transact-SQLCopy

SELECT \*

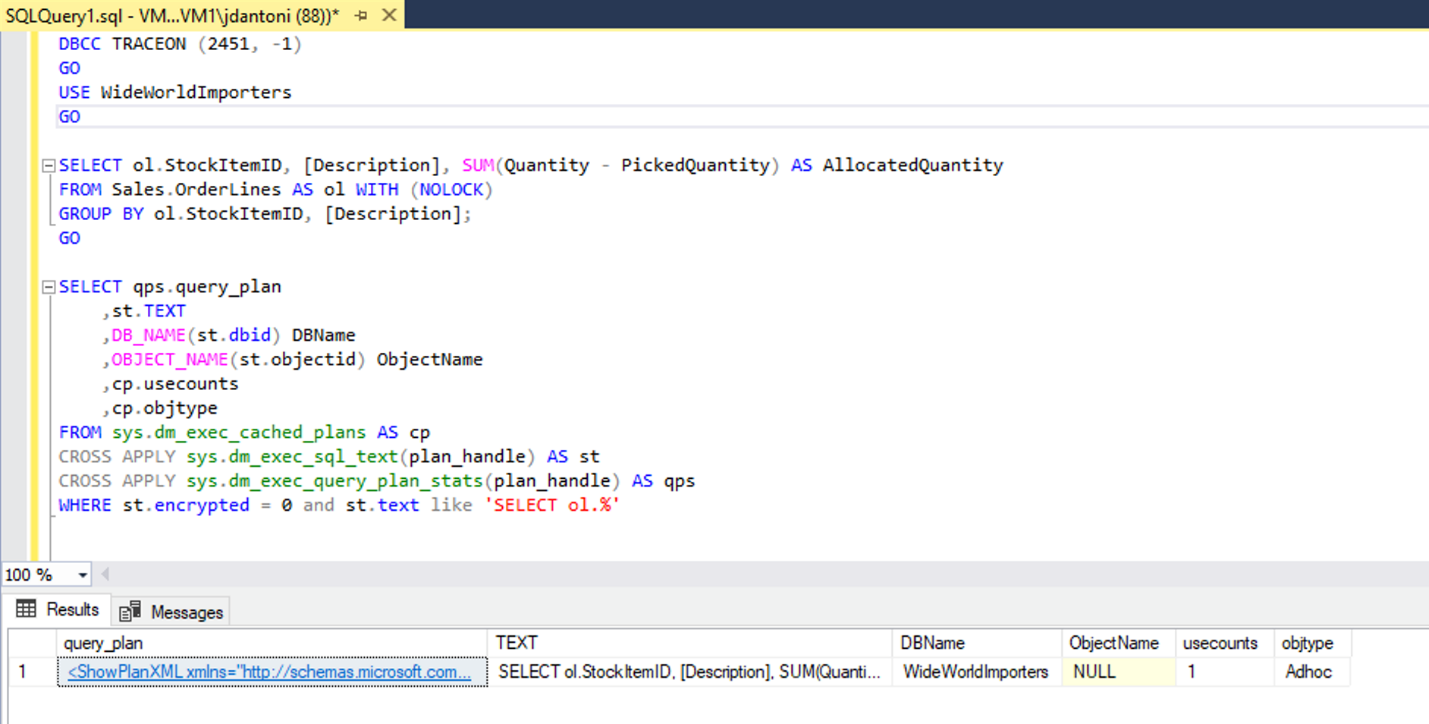
FROM sys.dm\_exec\_cached\_plans AS cp

CROSS APPLY sys.dm\_exec\_sql\_text(plan\_handle) AS st

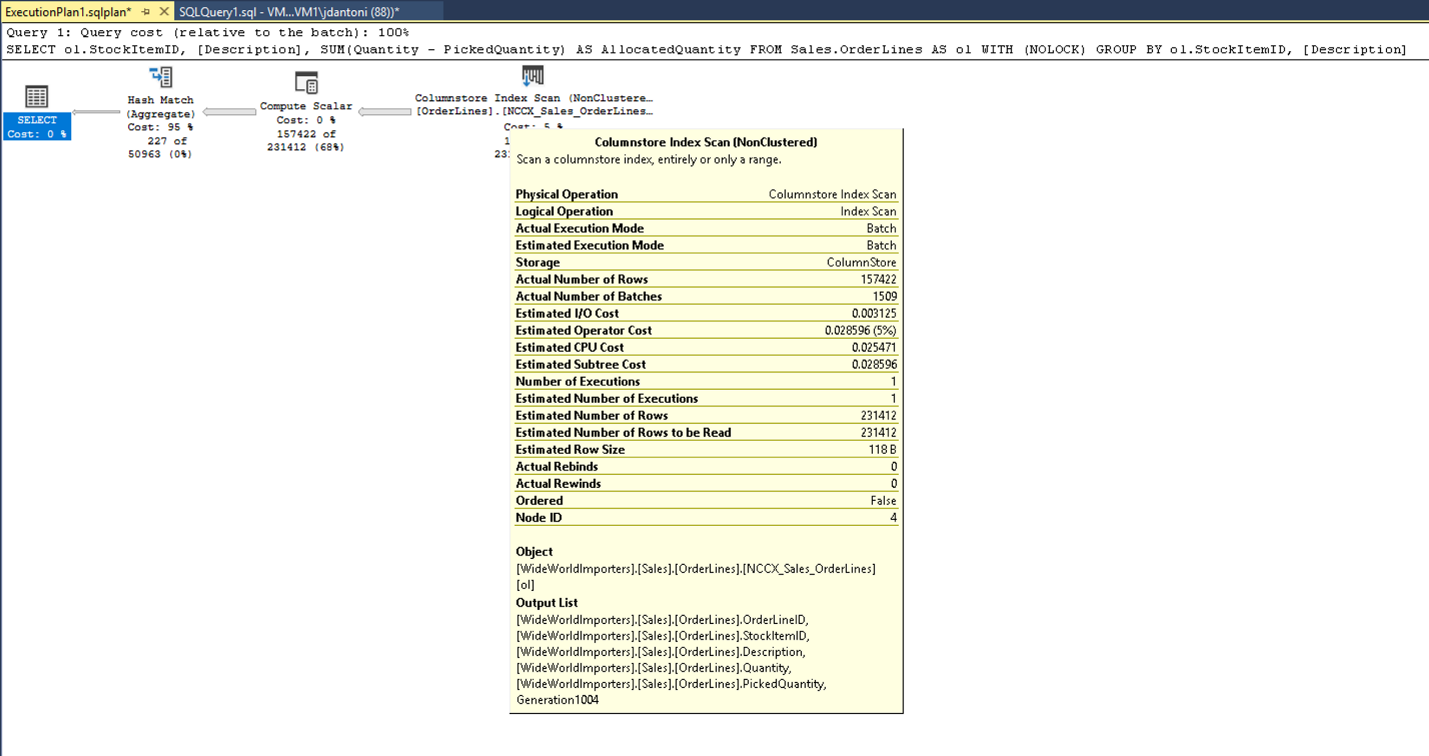
CROSS APPLY sys.dm\_exec\_query\_plan\_stats(plan\_handle) AS qps;

GO

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



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

[](https://learn.microsoft.com/en-us/training/wwl-data-ai/explore-query-performance-optimization/media/module-55-optimize-queries-final-06.png#lightbox)

**Describe dynamic management views and functions**

Completed100 XP

* 3 minutes

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](https://learn.microsoft.com/en-us/azure/azure-sql/database/monitoring-with-dmvs).

**Note**

For older versions of SQL Server where the query store is not available, you can use the view sys.dm\_exec\_cached\_plans in conjunction 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 will not 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 below 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.

# Explore Query Store

Completed100 XP

* 13 minutes

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

In total, the Query Store contains three stores:

* Plan store - used for storing estimated execution plan information
* Runtime stats store - used for storing execution statistics information
* Wait stats store - for persisting 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:

SQLCopy

-- 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 many stages. Within 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 recompiles. If the recompile results in a newly generated execution plan, it also persists in the Query Store to augment the previous plans. In addition, 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](https://learn.microsoft.com/en-us/sql/relational-databases/performance/how-query-store-collects-data).

## Common scenarios

The SQL Server Query Store provides valuable insight into the performance of the operations performed in a database. The most common scenarios include:

* Identifying and fixing performance regression 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
* Audit the history of query plans for a query
* Identifying and improving ad hoc workloads
* Understand the prevalent wait categories of a database and the contributing queries and plans affecting wait times
* Analyze database usage patterns over time as it applies to 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 is a query that is experiencing performance degradation over time due to execution plan changes. Estimated execution plans change due to many factors, including schema changes, statistics changes, and index changes. The first instinct may be to investigate the procedure cache, but the problem with the procedure cache is that it only stores the latest execution plan for a query; even then, plans are evicted based on the memory demands of the system. However, the Query Store persists several execution plans stored for each query, thus providing the flexibility to choose a specific plan in a concept known as plan forcing to solve the issue of a query performance regression caused by a plan change.

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

The plan summary pane displays the persisted query plans associated with the query over time after selecting a query from the top-left query pane. You'll see a graphical query plan in the bottom pane by selecting a query plan in the Plan Summary pane. In addition, toolbar buttons are available in both the plan summary pane and graphical query plan pane to force the selected plan for the selected query. This pane structure and behavior is consistently used across all SQL Query views.

Alternatively, you can use the sp\_query\_store\_force\_plan stored procedure to use plan forcing.

SQLCopy

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 ad hoc nature of the workload when identifying and improving ad hoc 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 tuning, 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](https://learn.microsoft.com/en-us/sql/relational-databases/system-catalog-views/sys-query-store-wait-stats-transact-sql).

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 will drill 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 will display 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.

# Identify problematic query plans

Completed100 XP

* 15 minutes

The path most DBAs take to troubleshoot query performance is to first identify the problematic query (typically the query consuming the highest amount of system resources), and then retrieve that query’s execution plan. There are two scenarios. One is that the query consistently performs poorly. Consistent poor performance can be caused by a few different problems, including hardware resource constraints (though this situation typically won't affect a single query running in isolation), a suboptimal query structure, database compatibility settings, missing indexes, or poor choice of plan by the query optimizer. The second scenario is that the query performs well for some executions, but not others. This problem can be caused by a few other factors, the most common being data skew in a parameterized query that has an efficient plan for some executions, and a poor one for other executions. The other common factors in inconsistent query performance are blocking, where a query is waiting on another query to complete in order to gain access to a table, or hardware contention.

Let’s look at each of these potential problems in more detail.

## Hardware constraints

Usually, hardware constraints won't manifest themselves with single query executions but will be evident when production load is applied and there's a limited number of CPU threads and a limited amount of memory to be shared among the queries. When you have CPU contention, it will usually be detectable by observing the performance monitor counter ‘% Processor Time’, which measures the CPU usage of the server. Looking deeper into SQL Server, you may see SOS\_SCHEDULER\_YIELD and CXPACKET wait types when the server is under CPU pressure. However, in some cases with poor storage system performance, even single executions of a query that is otherwise optimized can be slow. Storage system performance is best tracked at the operating system level using the performance monitor counters ‘Disk Seconds/Read’ and ‘Disk Seconds/Write’ which measure how long an I/O operation takes to complete. SQL Server will write to its error log if it detects poor storage performance (if an I/O takes longer than 15 seconds to complete). If you look at wait statistics and see a high percentage of PAGEIOLATCH\_SH waits in your SQL Server, you might have a storage system performance issue. Typically, hardware performance is examined at a high level, early in the performance troubleshooting process, because it's relatively easy to evaluate.

Most database performance issues can be attributed to suboptimal query patterns, but in many cases running inefficient queries will put undue pressure on your hardware. For example, missing indexes could lead to CPU, storage, and memory pressure by retrieving more data than is required to process the query. It's recommended that you address suboptimal queries and tune them, before addressing hardware issues. We’ll start looking at query tuning next.

## Suboptimal query constructs

Relational databases perform best when executing set-based operations. Set-based operations perform data manipulation (INSERT, UPDATE, DELETE, and SELECT) in sets, where work is done on a set of values and produces either a single value or a result set. The alternative to set-based operations is to perform row-based work, using a cursor or a while loop. This type of processing is known as row-based processing, and its cost increases linearly with the number of rows impacted. That linear scale is problematic as data volumes grow for an application.

While detecting suboptimal use of row-based operations with cursors or WHILE loops is important, there are other SQL Server anti-patterns that you should be able to recognize. Table-valued functions (TVF), particularly multi-statement table-valued functions, caused problematic execution plan patterns prior to SQL Server 2017. Many developers like to use multi-statement table valued functions because they can execute multiple queries within a single function and aggregate the results into a single table. However, anyone writing T-SQL code needs to be aware of the possible performance penalties for using TVFs.

SQL Server has two types of table-valued functions, inline and multi-statement. If you use an inline TVF, the database engine treats it just like a view. Multi-statement TVFs are treated just like another table when processing a query. Because TVFs are dynamic and as such, SQL Server doesn't have statistics on them, it used a fixed row count when estimating the query plan cost. A fixed count can be fine, if the number of rows is small, however if the TVF returns thousands or millions of rows, the execution plan could be inefficient.

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

## SARGability

The term SARGable in relational databases refers to a predicate (WHERE clause) in a specific format that can use an index to speed up execution of a query. Predicates in the correct format are called ‘Search Arguments’ or SARGs. In SQL Server, using a SARG means that the optimizer will evaluate using a nonclustered index on the column referenced in the SARG for a SEEK operation, instead of scanning the entire index (or the entire 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 was too expensive. This could be the case if a SARG refers to a large percentage of rows in a table. The absence of a SARG does mean that the optimizer won't even evaluate a SEEK on a nonclustered index.

Some examples of expressions that aren't SARGs (sometimes said to be non-sargable) are those that include a LIKE clause with a wildcard at the beginning of the string to be matched, for example, WHERE lastName LIKE ‘%SMITH%’. Other predicates that aren't SARGs occur when using functions on a column, for example, WHERE CONVERT(CHAR(10), CreateDate,121) = ‘2020-03-22’. These queries with non-sargable expressions are typically identified by examining execution plans for index or table scans, where seeks should otherwise be taking place.

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 much smaller set of rows, as shown below.

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

**Note**

The LIKE keyword, in this instance, does not have a wildcard on the left, so it is 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 1267 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 we see as database administrators are due to a lack of useful indexes causing the engine to read far more pages than necessary to return the results of a query. While indexes aren't free in terms of resources (adding more indexes to a table can affect write performance and consume space), the performance gains they offer can offset the extra resource costs many times over. Frequently execution plans with these performance issues can be identified by the query operator Clustered Index Scan or the combination of the Nonclustered Index Seek and Key Lookup (which is more indicative of missing columns in an existing index).

The database engine attempts to help with this problem by reporting on missing indexes in execution plans. The names and details of the recommended indexes are available through a dynamic management view called sys.dm\_db\_missing\_index\_details. There are also other DMVs in SQL Server like sys.dm\_db\_index\_usage\_stats and sys.dm\_db\_index\_operational\_stats, which highlight the utilization of existing indexes.

It may make sense to drop an index that isn't used by any queries in the database. The missing index DMVs and plan warnings should only be used as a starting point for tuning your queries. It’s important to understand what your key queries are and build indexes to support those queries. Creating all missing indexes without evaluating indexes in the context of each other isn't recommended.

## Missing and out-of-date statistics

You've learned about the importance of column and index statistics to the query optimizer. It's also important to understand conditions that can lead to out-of-date statistics, and how this problem can manifest itself in SQL Server. Azure SQL offerings default to having auto-update statistics set to ON. Prior to SQL Server 2016, the default behavior of auto-update statistics was to not update statistics until the number of modifications to columns in the index was equal to about 20% of the number of rows in a table. Because of this behavior, you could have data modifications that were significant enough to change query performance, but not update the statistics. Any plan that used the table with the changed data would be based on out-of-date statistics and would frequently be suboptimal.

Prior to SQL Server 2016, you had the option of using trace flag 2371, which changed the required number of modifications to be a dynamic value, so as your table grew in size, the percentage of row modifications that was required to trigger a statistics update got smaller. Newer versions of SQL Server, Azure SQL Database and Azure SQL Managed Instance support this behavior by default. There's also a dynamic management function called sys.dm\_db\_stats\_properties, which shows you the last time statistics were updated and the number of modifications that have been made since the last update, which allows you to quickly identity statistics that might need to be manually updated.

## 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 may 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 will be predictable. The value of ‘Seattle’ for @city\_name will only be used during optimization. During execution, the actual supplied value (‘Ascheim’) will be used.

SQLCopy

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 example below:

SQLCopy

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 usually 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 will optimize 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 will be 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 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 will cause 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 will cause 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 will be executed with the OPTION (RECOMPILE).

SQLCopy

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 WHERE SalesPersonID = @SalesPersonID'

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

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

# Describe blocking and locking

Completed100 XP

* 12 minutes

One feature of relational databases is locking, which is essential to maintain the atomicity, consistency, and isolation properties of the ACID model. All RDBMSs will block actions that would violate the consistency and isolation of writes to a database. SQL programmers are responsible for starting and ending transactions at the right point, in order to ensure the logical consistency of their data. In turn, the database engine provides locking mechanisms that also protect the logical consistency of the tables affected by those queries. These actions are a foundational part of the relational model.

On 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 when the process holding the lock releases it, the blocked process can then acquire the lock and complete its transaction.

SQL Server locks the smallest amount of data needed to successfully complete the transaction. This behavior allows maximum concurrency. For example, if SQL Server is locking a single row, all other rows in the table are available for other processes to use, so concurrent work can go on. However, each lock requires memory resources, so it’s not cost-effective for one process to have thousands of individual locks on a single table. SQL Server tries to balance concurrency with cost. One technique used is called lock escalation. If SQL Server needs to lock more than 5000 rows on a single object in a single statement, it will escalate the multiple row locks to a single table lock.

Locking is normal behavior and happens many times during a normal day. Locking only become a problem when it causes blocking that isn't quickly resolved. There are two types of performance issues that can be caused by blocking:

* A process holds locks on a set of resources for an extended period of time before releasing them. These locks cause other processes to block, which can degrade query performance and concurrency.
* A process gets locks on a set of resources, and never releases them. This problem requires administrator intervention to resolve.

Another blocking scenario is deadlocking, which occurs when one transaction has a lock on a resource, and another transaction has a lock on a second resource. Each transaction then attempts to take a lock on the resource, which is currently locked by the other transaction. Theoretically, this scenario would lead to an infinite wait, as neither transaction could complete. However, the SQL Server engine has a mechanism for detecting these scenarios and will kill one of the transactions in order to alleviate the deadlock, based on which transaction has performed the least of amount of work that would need 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. Auto-commit is the default mode of SQL Server and Azure SQL Database, which means the changes made by the statement below would automatically be recorded in the database's transaction log.

SQLCopy

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 query below would take a lock on a row in the DemoTable table what wouldn't be released until a subsequent command to commit the transaction was added.

SQLCopy

BEGIN TRANSACTION

INSERT INTO DemoTable (A) VALUES (1);

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

SQLCopy

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 will stay 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:

SQLCopy

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 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\_process

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 with 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:

SQLCopy

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 problem can be troublesome as they can be sporadic in nature. There's a DMV called sys.dm\_tran\_locks, which can be joined with sys.dm\_exec\_requests in order to provide further information on locks that each session is holding. A better way to monitor for blocking problems is to do so on an ongoing basis using the Extended Events engine.

Blocking problems typically fall into two categories:

* Poor transactional design. As shown above, a transaction that has no COMMIT TRANSACTION will never end. While that is a simple example, trying to do too much work in a single transaction or having a distributed transaction, which uses a linked server connection, can lead to unpredictable performance.
* Long running transactions caused by schema design. Frequently this can be an update on a column with a missing index, or poorly designed update query.

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

For more information about how to monitor blocking, see [Understand and resolve SQL Server blocking problems](https://learn.microsoft.com/en-us/troubleshoot/sql/performance/understand-resolve-blocking).