Execution Plans

Last updated by | Vitor Tomaz | Feb 24, 2023 at 3:32 AM PST

Contents

What is an execution plan

- steps that are taken by SQL to run a query
- An execution plan is compiled when a query is executed for the first time (or if the plan is no longer on the buffer cache).
- An execution plan is recompiled when statistics or schema changes occur.
- Since plan compilation is a CPU expensive operation, SQL will always try to reuse plans. In other words, plans aren`t always compiled (except in some ocasions. See below on Plan Compilation/Recompilation).
- When compiling, the optimizer produces multiple plans and chooses the one with less cost.
- Stored as xml.
- Plan compilation is done based os statistics. Like so, good statistics generally produces better plans.

Types of execution plans

- *Estimated*: plan generated without running the query. To retrieve the estimated execution plan press Ctrl+L on SSMS and Azure Data Studio.

- *Actual*: the plan actually used by SQL. It`s given when running the query. To retrieve the actual execution plan press Ctrl+M on SSMS and Azure Data Studio.

Most of the times the estimated plan is the same as the actual, but note that the actual plan will give you more data, like Actual number of rows and live statistics (for example, the waits observed while running the query)

Actual execution plan different when executing from SSMS

In some cases the execution plan (and execution times) can be different when executing the same query on SSMS and on the application.

Check the TSG Different execution plan when executing from SSMS to have more details on what might affect the time and execution plan difference when executing from application vs SSMS.

How to read an execution plan

- the plan should be read from right to left, top to bottom
- each operator has a cost in % in terms of the overall plan (for example, you will always have an operator with high cost. That cost is for that plan only.)
- each operator is connected by arrows. The size of each arrow will represent the amount of data that is being exchanged between operators (the thicker the arrow is, the more data is being transferred).
- each single operator will contain metadata properties, like IO cost, number of rows read, etc.

Some interesting data to look on an execution plan

Cardinality estimator

This will be on the top of the execution plan and will tell you what was the cardinality estimator version used For example:

> CardinalityEstimationModelVersion="130"

The cardinatility estimator version usually will be tied to the Database compatility level. In some ore rare cases to a query hint

MAXDOP and memory grant

This will tell you if a plan used parallelism, and if it´s not using parallelism will tell you why. Also will contain information about memory grants

Example:

> OptimizerHardwareDependentProperties EstimatedAvailableMemoryGrant="52428" EstimatedPagesCached="65536" EstimatedAvailableDegreeOfParallelism="10" MaxCompileMemory="6668440"

Plan compilation result

It will tell if it`s a Trivial Plan (simple queries will use Trivial plans), if it Timeout, or if the plan compilation was completed (marked as FULL)

> StatementOptmLevel="FULL"

On a case of Execution Plan Timeout, you might want to check https://techcommunity.microsoft.com/t5/sql-server-support-blog/understanding-optimizer-timeout-and-how-complex-queries-can-be/ba-p/319188

Warnings

This will contain some interesting data regarding some warning events, like implicit conversions or other events:

Example:

> ConvertIssue="Cardinality Estimate" Expression="CONVERT_IMPLICIT(nvarchar(max),[Union1353],0)

Missing indexes

Information on missing indexes. Note that, if opening the execution plan with SSMS, the graphical interface will only show you one missing index. To see all the missing indexes you have to check the plan xml or use another tool like SQL Sentry Plan Explorer.

Example:

> MissingIndexGroup Impact="99.1552" MissingIndex Database="[database_name]" Schema="[dbo]" Table="[TableName]" ColumnGroup Usage="EQUALITY" Column Name="[Column1]" ColumnId="3" / Column Name="[Column2]" ColumnId="11" / /ColumnGroup ColumnGroup Usage="INCLUDE" Column Name="[Column3]" ColumnId="4" /

Wait Stats

On a actual execution plan, will tell you what was the type and duration of each type

> Wait WaitType="LATCH_EX" WaitTimeMs="4132" WaitCount="2124" / Wait WaitType="CXSYNC_PORT" WaitTimeMs="555" WaitCount="9" /

Statistics

Will tell you what was the state of the statistics when the plan was compiled. Look for low sampling values and last update date and hour

Example:

> StatisticsInfo Database="[database]" Schema="[dbo]" Table="[TableName]" Statistics="[statistics_name]" ModificationCount="2714757" SamplingPercent="3.99437" LastUpdate="2022-09-20T05:52:11.13" /

Operators

The list of available operators is quite extensive. They are listed on https://learn.microsoft.com/en-us/sql/relational-databases/showplan-logical-and-physical-operators-reference?view=sql-server-ver16

Some common misconceptions around some operators:

- *Table (or index) scans are always bad* - Not necessarily true. For example, if the table is small (for example, if the records fit in one or two data pages, the cost of the scan will be small).

- *Seeks are preferred* - It will depend on the amount of data that is being retrieved. For example, if you are retrieving 95% of a table, a SCAN will be preferred
- *A key lookup is always bad* - not always. If the number of rows that is being retrieved through the nested loop join is small, the cost of having a covered index (that sometimes can require a lot of rows) is more than retrieving one or two records from the key lookup.

JOIN algorithms

Understanding the different JOIN algorithms is key to understand some problems like Tempdb and performance issues. Explaining below some of the most common join algorithms

Nested Loop

For each row from the outer table, performs a search on the inner table for the corresponding record. It will use less memory when compared with other JOINs. It`s efficient when the Outer table returns a small amount of data and the search on the inner table is optimized (index seek). In short, will be a loop (on the inner table) inside another loop (on the outer table).

Nested loops are better when the number of rows is small and when the inner table access is done on a efficient manner.

Hash Match

In short, creates a hash table from one data set and then compares each row to find matching records.

- First there will be a build phase, where a hash table will be created in memory using one of the tables (usually the smallest one). Then the hashes will be calculated using a hash function.
- Then we will have a Probe phase, where the join key hash of the other table will be calculated. Then it will check if the there`s a match for the hash.

Some remarks regarding Hash Joins

- The hash table is built in memory. Like so, if there`s an incorrect estimate in the memory grant, or there isnt enough memory, it will spill to tempdb
- Usually, the right choice when we have two large non sorted inputs.
- the Build phase is a blocking operation.
- when troubleshooting tempdb problems, Hash match operators on execution plans are a good place to look at (among other operators).

Merge Join

Both data sets are read and merged. It requires sorted data. If the data isn`t sorted (data is already sorted on an index), SQL will Sort before performing the Merge Join. Since the data is sorted, is very efficient algorithm.

In other words, it uses the data already sorted on an index to perform the Join. If the data is not sorted, the optimizer can choose this algorithm if the sorting operation is not expensive.Merge Join can become a problem when the Sort operation has a high cost.

Merge Join is good on One-to-Many relationships. Many-to-many has a performance hit.

On a situation where statistics are out of date (incorrect estimates), the optimizer can assume that the Merge Join is good option because the estimate of the Sort operation is low.

When looking at an execution plan, check if there is a Sort operator just before the Merge Join. If yes, check the number of rows that are being sorted. Existence of tempdb spills on the sorting might also be a good indication of bad estimates.

Update statistics to fix any problem with estimates.

What to look on an execution plan

Some points on what to look quickly, before a deep dive:

- look for missing indexes. Usually, they are a good starting point. But just don`t blindly create them (sometimes already existing indexes can be reused).
- Look for implicit conversions on columns. No indexes can really help when variables aren't correctly declared
- Look for warning on the plan (for example, spills to tempdb)
- Take a brief look at the plan and think if the problem isn't on the query logic (for example, a row-by-row execution)
- Outdated statistics is one common problem. Look at Estimated vs Actual rows and look for big differences.

How good have you found this content?