**SQL Server Query Plan Analysis: The 5 Culprits That Cause 95% of Your Performance Headache**

[SQL Server Query Plan Analysis: The 5 Culprits That Cause 95% of Your Performance Headache (youtube.com)](https://www.youtube.com/watch?v=bS0q1nBP3As)

A green and orange square with black text

Description automatically generated

**From simple plan to complex plan**

Estimated vs actual execution plan: the plan is 99.9% same, except recompile is involved.

Select \* from production.product

Hover on the “select” on the plan (this is estimated plan)

A screenshot of a computer

Description automatically generated

A screenshot of a computer screen

Description automatically generated

Est number of execution: 1 means the query (select operator) will be executed once

Est number of rows: # of rows that will be fetched

Est row size: size of the rows fetched

A screenshot of a computer screen

Description automatically generated

Cost is est number of seconds that the query is expected to run in old machine 1996.

Almost meaningless?? Don’t pay attention on it. Focus on other things.

Actual plan

A screenshot of a computer

Description automatically generated

See the “Actual” number of rows and number of execs.

Actual cost is not reflected in actual plan. They are all estimated costs.

Run

Select top (100) from production.product

A screen shot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

It has Index scan…but Actual number says 100.

THIS MEANS **INDEX SCAN DOESN’T NECESSARILY SCAN THE WHOLE TABLE.**

**HOW THE EXECUTION PLAN WORKS: how to properly read it**

A screen shot of a computer

Description automatically generated

In the above example

A screen shot of a computer

Description automatically generated

Select – top – cluster index scan : are called **iterators** – iterators reiterates (doing something over and over)

Select will say to top do you have rows, top will say no le me ask cluster index scan – cluster index scan will bring him a row – and does it until the top’s requested rows 100 times. Then cluster index scan shuts down, then top shuts down and select will shutdown and plan ends.

New example

At the end

-Actual cost

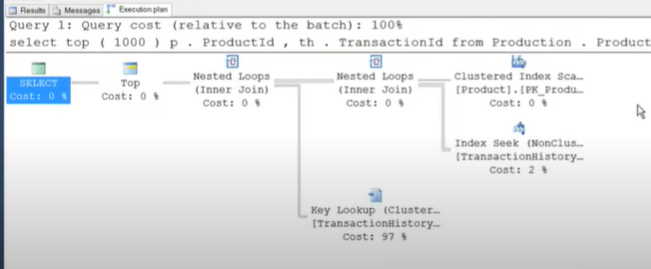
-p.standardcost

are predicates

A screenshot of a computer

Description automatically generated

The plan is as follows, followed by the reading and identifying the problem.



The first “cluster index scan” show the rows it scanned as 223 on product table. But this wasn’t the only table called for, so there is “index seek” which brought 40059 rows by doing 223 index operations against the transaction history table with nested loop (for each row scanned from the first CIS).

A screenshot of a computer screen

Description automatically generatedA screenshot of a computer screen

Description automatically generated

So far so good as long as the execution in lower (in this case upto 223).

But, the problem lies with the next operator (“**key lookup**”). It does 40059 operations/lookups. This will need tuning.

A screenshot of a computer

Description automatically generated

Summary

A screen shot of a chart

Description automatically generated

**Lines**:

show where the actions and rows are concentrated**. Thick more** rows, **thin fewer** rows. Make lines thinner as it passes to the right (select).

A screenshot of a computer

Description automatically generated

Views on top of view will make the optimizer not to make the line thinner. Makes the query super slow.

**Solution**

Use Table-valued function (inline) and replace views. You can push the line thinner to the right with table valued function, the optimizer doesn’t have to do it.

**TOP FIVE “QUICK CHECK” ITERATOR**

Start with THICKER LINES.

THEN LOOK THE FOLLOWING 5 CULPRITS.

1. LOOKUP
2. SPOOL
3. SORT
4. HASH
5. NESTED LOOPS (SERIAL)
6. BONUS – SCAN

A screen shot of a computer screen

Description automatically generated

**LOOKUP**:

RID LOOKUP (HEAP) AND KEY LOOKUP (CLUSTERED)

Lookup for small number of rows is okay, but for large- it is a problem.

Happens when we have filter but one or more of columns are not included in the index.

A blue and orange squares with white text

Description automatically generated

**SPOOLS**

Is A ***Cache*** in The Query Processor. Happens For **2 Reasons**: Lack Of ***Adequate Index*** Or ***Uniqueness Of Information (PK, unique key..)***. Sometimes Even if you Have the Index, the **Optimizer** Can ***Forget it*** due to the behaviour of various iterators and you can **remind it by adding** “**Distinct**”.

Is Like a **Hidden Table In Tempdb** – **Never A Good Sign**. Queries it and move the data to tempdb, queries it and move the data to tempdb…again and again.

A blue and orange rectangular box with white text

Description automatically generated

***Lazy spool = performance spool***

**SORT**

Order by, merger join, stream agg, windowing/window functions

Performs worse, relative to input size, as input size increases.

Too much sort in the plan is not good.

App can do it, instead of sql server. It is costly when done with SQL server.

A blue and orange squares with white text

Description automatically generated

**HASH**

Hash match – for join – uses hash table.

Hash match (aggregate) – for aggregation.

Has match built a hash table in memory, if no enough space – it will build it in tempdb.

**Common in Warehouses**, not OLTP. If it happens in OLTP, it shows **failure to do index**.

A blue and orange rectangular box

Description automatically generated

**Nested Loops (serial)**

For joins. Small outer (top) input.

One day 5 seconds and the next day **it hung for hours**. This can be due to invalid nested loops (serial).

**You have 10000 rows, but SQL optimizer thinks you have 1 or 0 rows.**

A blue and orange square with white text

Description automatically generated

**SCAN**

* start from the top and read down as many to

Table Scan

Index Scan (Non-Clustered)

Clustered Index Scan (Clustered)

Not Quite Often A Problem.

A blue and white rectangular box with white text

Description automatically generated

**DEMO**

**LOOKUP**

* MAKE the situation as bad as it can be. (gets read of all the pages in buferpool that haven’t been modified). NEVER DO IT ON PROD.

*DBCC DROPCLEANBUFFERS*

*GO*

(gets read of all the pages in buferpool that haven’t been modified).

A screenshot of a computer

Description automatically generated

Read this. (this is a 0 second operation)

A screenshot of a computer

Description automatically generated

Reading: It does a ***cluster index scan*** on bigprodcut table then **loops** to bigtransaction table (do ***index seek*** on matches) and selects the ***top*** 100 from there then does gather ***streaming*** then does an ***insert*** to a temp table.

Now a new row was added to a table bigtransactionhistory. But the previous select query remains the same BUT THE PLAN CHANGED A LOT. READ IT.

A blue rectangle with black text

Description automatically generated

NEW PLAN

A screenshot of a computer

Description automatically generated

READING: **index scan** on *bigproduct* table and **loops** to bigtransaction table to do matching. it does ***index seek*** on bigtransaction and ***sorted*** the data/returned rows/ to ease the ***lookup*** to another column. Then result is **sorted** **with TOP N** sort, then ***streams*** the gatherd data, finally does table ***insert***.

The query now takes 17 seconds from 0 seconds.

3 main reason to cause this:

1. Developers work in an environment that doesn’t have as much data as prod. Must be tested in reasonable size.
2. Lack of testing in general, LOAD TEST.

**Outer most** select star in stead of select \* - this is an issue.

A blue and green text

Description automatically generated

There is **a suggested index** in the above plan, **but be careful in applying it**. For instance, here **we have an index already on Product ID that INCLUDE every column but [customerid]**…so in stead of creating new index…Just **UPDATE the existing INDEX.** Instead of duplicating an index.

A computer screen shot of a missing link

Description automatically generated with medium confidence

**SPOOLS**

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated A screenshot of a computer code

Description automatically generated

the plan

A screenshot of a computer

Description automatically generated

Reading: data is take from #products table and the input is sorted (sorted on productid, so that all prodid are grouped together), then we have lazy spool/performance spool. Each sorted row goes into table spool and ask do you have any information about this productid (e.g. if the first prodid is 1, do you have info about prodid=1?). lazyspool goes to bigtransactionhistorytable (run the sub-query) and gets the information and sorts it and puts in tempdb and forward the data.

Lazy spool After bring the information, it forward the information whenever asked…since it already store it as sorted it doesn’t have to go to bigtrxnhistory table again and again. The sorting is costly, it is good that it is already saved as sorted.

A screenshot of a computer

Description automatically generated

To see further information hover over the spool.

**Actual rebinds**: when the subtree behind this spool has to get re-run

**Actual rewinds**: data is served straight from this pool.

Number of execution (rebind + rewind): how many times data was requested from this spool. 12845 time. From this we run the subtree 9577 times and we serve the data 3268 times. We can see this from one back from the spool operator .

A screenshot of a computer

Description automatically generatedA yellow line on a white background

Description automatically generated



This is bad, needs spool canceled.

**2 ways to avoid it**

1. Add “select distinct”

A screen shot of a computer

Description automatically generatedA screen shot of a computer

Description automatically generated

Don’t do it on prod but you can check what happens with **trace flag 8690**– it will get rid of spool. Lowers it to 3 sec from 17. Just test only in non-prod.

A screenshot of a computer code

Description automatically generated

**SORT**A screenshot of a computer

Description automatically generatedA computer screen shot of a code

Description automatically generated

Shows as products with highcost (>1500). Calculate row number, with every product we will find a set of row numbers.

The plan

A screenshot of a computer

Description automatically generated

It is **sorting 148mill rows**. That is the highest cost. The index suggestion has minimal benefit (15% only). The query ran for 6 seconds.

There is a yellow exclamation as well. It tells that the sorting was beyond memory, and used the disk for sorting.

This is the key in the query

A close up of words

Description automatically generated

Cross apply – almost always optimizes as a nested loop.

**Solution: Making it do smaller sorts == better performance**.

A screen shot of a computer

Description automatically generated

The new plan (down from 6 to 3sec). the sort looks a bit expensive (73%, prev was 71%) but better. It is better to do much small sorts than one monolithic sort.

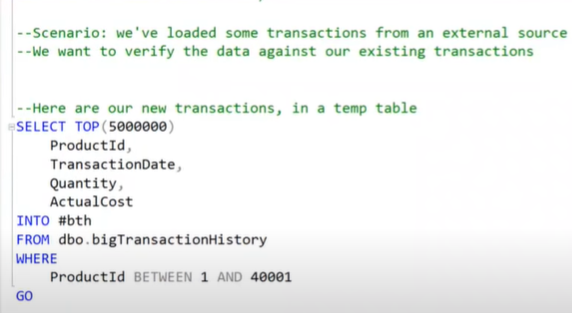
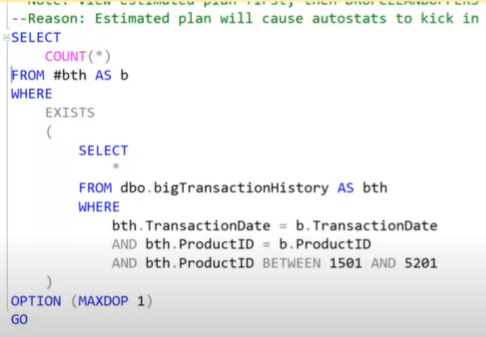
A screenshot of a computer

Description automatically generated

It is always beneficial for SQL server to do somany things in small amount instead of doing it one monolithic big thing.

**HASH MATCH**

* Not good for OLTP, but good for warehouse
* Good when we have matching one medium size and on bigger sets of rows.
* **Happens when we don’t have an appropriate index**.

Counting products between 1501 to 5201.

Running the second part (count (\*))

A screenshot of a computer

Description automatically generated

Index seek was done on bigtransactionhistory table on smaller set of product – because we don’t have index in the #bth table. So SQL server chose to do hash match. This plan shows as to consider indexing the temp table #bth )12% improvement .

A screenshot of a computer

Description automatically generated

It suggests index as follows

Create cluster index (name ..e.g. ix\_productID\_transactionDate)

On #bth ([productid])

Include ([transactiondate])

Go

We can similar one

Create cluster index ix\_productID\_transactionDate

On #bth

(productid,

Transactiondate)

go

A computer screen shot of text

Description automatically generated

New plan: avoids hash match and does merge join.

A screenshot of a computer

Description automatically generated

If you recompile it you can see it can improve 100% instead.

Remember:

* **when you build or rebuild an index, it also does a full update statistics. Don’t create stat on indexes after building a new index (you may come up with bad stat).**
* **Don’t do stat update with sampling (e.g. 20%) percentage after an index rebuild...you are dragging two steps back. It already does full stat update.**

**Estimated 1, actual 3744403**

**The problem is histogram for statistics. Ascending key problem logic.**

**This is solved in 2014 and later? Traceflag 2371 also solves?**

A screenshot of a computer

Description automatically generated

**SCAN**

For checking logical, physical read

*Set statistics IO ON*

*Go*

This query

A screenshot of a computer

Description automatically generated

Has plan as follows…it looks simple but something is not right. It shows there is residual predicate in addition to seek predicate. Residual predicate will go to the disk.

There is no output on this (0 row turned) because actual cost > 50000 doesn’t exist.

A screen shot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

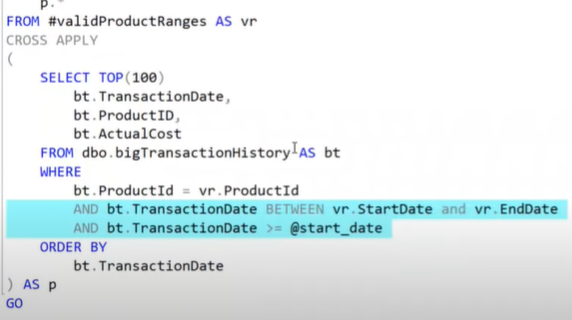
There is no thick line to see where the problem lies. Use a TF (trace flag) 9130 to test (never on prod) where the data is flowing:

A computer screen shot of a number

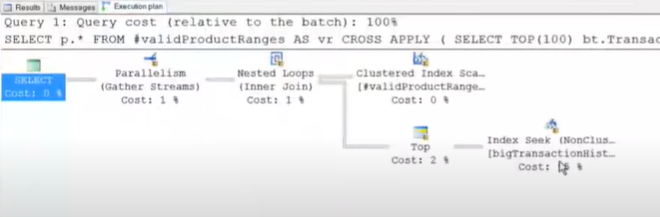
Description automatically generated A screenshot of a computer

Description automatically generated

Another example: involves two predicates startdate amd emd date – in such cases SQL SERVER has to choose one predicate (seek or residual)



The plan



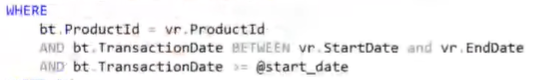
Start and end date in a table as seek predicate. And start date as residual

A screenshot of a computer

Description automatically generated

**Solution: use case expression**

Instead of doing transaction date between start and end date and start date



Use case expression!

A screenshot of a computer

Description automatically generated

Summary

A green and blue squares with white text

Description automatically generated

[SQL Server Performance Tuning Made Easy - Pinal Dave | Craft 2019 (youtube.com)](https://www.youtube.com/watch?v=SqhX8OaOI6A)

Pinal here

Index that is not used can cause slow performance.