SQL Performance tuning

12 step program

Brian Bønk Rueløkke
Principal Architect

Fellowmind

Introduction

Query tuning is a powerful tool for DBAs and developers alike in improving SQL Server performance. Unlike measures of system-level server performance (memory, processors, and so on), query tuning puts the focus on reducing the amount of logical I/O in a given query, because the fewer I/Os, the faster the query. In fact, some performance issues can only be resolved through query tuning, and focusing on system resources can lead to expensive and unnecessary hardware investments that don't in the end make the query faster.

Yet many DBAs struggle with query tuning. How do you assess a query? How can you discover flaws in the way a query was written? How can you uncover hidden opportunities for improvement? How can you be certain that making a specific alteration actually improves the speed of the query? What makes query tuning as much an art as a science is that there are no right or wrong answers, only what is most appropriate for a given situation.

This paper demystifies query tuning by providing a rigorous 12-step process that database professionals at any level can use to systematically assess and adjust query performance, starting from the basics and moving to more advanced query tuning techniques like indexing. When you apply this process from start to finish, you will improve query performance in a measurable way, and you will know that you have optimized the query as much as is possible.

ALWAYS START WITH THE BASICS OF QUERY ANALYSIS

When asked to fix a slowly running query, experienced DBAs frequently skip directly to examining execution plans, and then run the query only to be surprised at how long it takes to return data. At that point, there is sometimes a realization that the table is really quite large. This is why I always advise to start with the basics—knowing exactly what you're dealing with before you dive in.

1.1 Know your tables and row counts

First, make sure you are actually operating on a table, not view or table-valued function.
If it's a view, you need the view definition. Table-valued functions have their own performance implications.
Hint: You can use SSMS to hover over query elements to examine these details.

```
SELECT
    COUNT(DISTINCT A.ModifiedDate) AS ModDateCount,
    A.ProductID,
    COUNT(A.CarrierTrackingNumber) AS CarrierCount
FROM SalesMaster A
WHERE view AdventureWorks2012.dbo.SalesMaster
    ModifiedDate >= @StartDate  AND
    ModifiedDate <= @EndDate AND
    A.ProductID <> 897
GROUP BY
    A.ProductID
```

Check the row count by querying the DMVs (see example below). If, for example, the query was built and tested in a development environment, but is being run in a production environment for the first time, the actual row count may be significantly higher.

```
SELECT so.name, sp.rows
FROM sys.objects so INNER JOIN sys.partitions sp
    ON so.object_id = sp.object_id
WHERE so.type IN ('TF', 'U', 'V')
AND sp.index_id = 1
ORDER BY so.name
```

	name	rows
46	PurchaseOrderDetail	8845
47	PurchaseOrderHeader	4012
48	Sales	485268
50	SalesOrderHeader	31465
51	SalesOrderHeaderS...	27647
52	SalesPerson	17
53	SalesPersonQuotaHi...	163

1.2 Examine the query filters.

Examine the WHERE and JOIN clauses and note the filtered row count

 Tip: If there are no filters, and the majority of table is returned, consider whether all that data is needed. If there are no filters at all, this could be a red flag and warrants further investigation. This can really slow a query down.

1.3 Know the selectivity of your tables

Based upon the tables and the filters in the previous two steps, know how many rows you'll be working with, or the size of the actual, logical set. We recommend Dan Tow's SQL Tuning for a robust discussion of selectivity and the use of SQL diagramming as a powerful tool in assessing queries and query selectivity.

This is important specifically for RIGHT, LEFT and OUTER joins. You should have a good understanding of when the predicate is applied so you can be sure you're starting with the smallest possible set and the filters are getting applied early enough.

1.4 Analyse the additional query columns - the extra things outside of the filters and JOINs

Examine closely the SELECT * or scalar functions to determine whether extra columns are involved.
Is there CASE, CAST, CONVERT happening in the WHERE clause? Is it SARGable (is the index searchable)? Are there sub-queries? The more columns you bring back, the less optimal it may become for an execution plan to use certain index operations, and this can, in turn, degrade performance.

CONTINUE ON TO MORE ADVANCED QUERY ANALYSIS

1.5 Review the existing keys, constraints, indexes to make sure you avoid duplication of effort or overlapping of indexes that already exist

Know and use these constraints because they can be very helpful as you start to tune.

What is the primary key definition? Is that key also clustered? If you have a wide clustered key, you must also copy the key over to your non-clustered indexes, which means more pages you have to read into the buffer pool to solve the query.
If you're not using foreign key constraints, consider whether they will work in your data model. The optimizer can make use of foreign key constraints to make better execution plans, which will help the query run faster.

To get information about your indexes, run the sp_helpindex stored procedure:

```
EXEC sp_helpindex 'Sales.Sales'
```

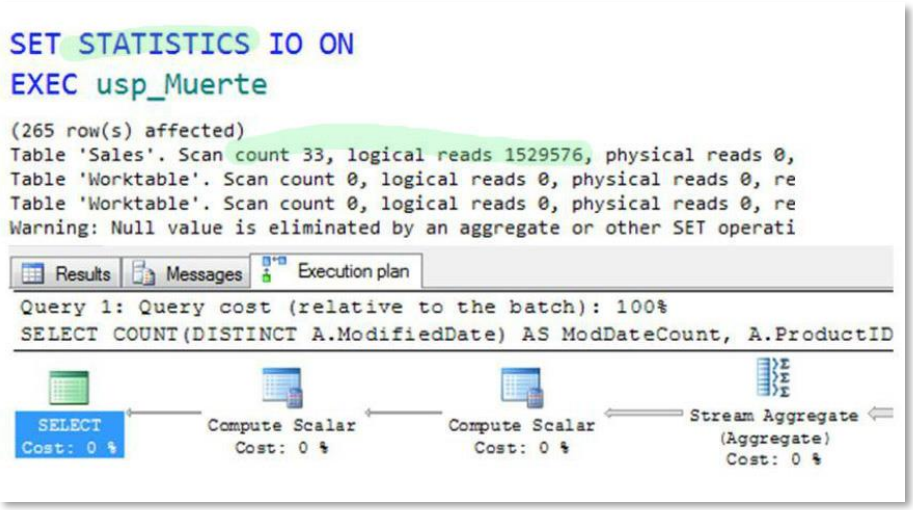
Results | Messages | Execution plan

	index_name	index_description	index_keys
1	AK_Sales_rowguid	nonclustered, unique, stats no recompute located o...	rowguid
2	cx_SalesSales	clustered, stats no recompute located on PRIMARY	SalesOrderID, SalesOrderDetailID
3	ix_modifiedSales	nonclustered, stats no recompute located on PRIMA...	ModifiedDate
4	IX_Sales_CarrierTrackingNumber	nonclustered, stats no recompute located on PRIMA...	CarrierTrackingNumber
5	IX_Sales_ProductID	nonclustered, stats no recompute located on PRIMA...	ProductID

Note, however, that the included columns are not included! If you need that information, you'll need to use a different query.

1.6 Examine the actual execution plan (not the estimated plan)

Estimated plans use estimated statistics to determine the estimated rows; actual plans use actual statistics at runtime. If the actual and estimated plans are different, you may need to investigate further.
Note that at this step, you will want to set statistics on (SET STATISTICS IO ON and SET STASTICIS TIME ON).



1.7 Record your results, focusing on the number of logical I/Os

This is an especially important step, and one that many DBAs will skip; if you don't record the results, you won't be able to determine the true impact of your changes alter on.

1.8 Adjust the query based on what you've found, making small, single changes one at a time

If you make too many changes at one time, you may find the changes cancel each other out! Begin by looking for the most expensive operations first. There is no right or wrong answer, but only what is optimal for the given situation (note that all of these will be affected by out-of-date statistics).
Some of the potentially expensive operations you might encounter include:

- Data transfers from one operation to the next. Is the actual number of rows much larger than
- the estimated? If estimates are off from actuals, it may indicate a need for further investigation.
- Are seeks or scans more expensive in this specific scenario? Contrary to common belief, a table scan may be less expensive than a seek, in some instances. For example, if the table is very small, SQL Server will read the whole table into memory regardless, and so a seek isn't necessary.

- Is parameter sniffing an issue (parameter sniffing results from re-using a previously cached plan that has been optimized for parameter values from the original execution, and those parameters may be very different)? Is it using local variables?
- Are there spool operations (a result set is stored in tempdb for use later), and if so, are they necessary?
- Which is better in the situation: LOOP, MERGE or HASH joins? It will depend on the specific circumstances and the statistics the optimizer is using.
- Are there lookups, and if so, are they necessary?

1.9 Re-run the query and record results from the change you made

If you see an improvement in logical I/Os, but the improvement isn't enough, return to step 8 to examine other factors that may need adjusting. Keep making one change at a time, rerunning the query and comparing results until you are satisfied that you have addressed all the expensive operations that you can.

1.10 If at this point you believe the query is written as well as it possibly could be and you still need more improvement, consider adjusting the indexes to reduce logical I/O

Adding or adjusting indexes isn't always the best thing to do, but if you can't alter the code, it may be the only thing you can do.

- Consider the existing indexes. Are they being used effectively? Focus on those tables with the lowest selectivity first.
- Consider a covering index—an index that includes every column that satisfies the query. Be sure to first examine the Delete/Update/Insert statements: what is the volume of those changes?
- Consider a filtered index (SQL Server 2008 and later)—a non-clustered index that has a predicate or WHERE clause. But be aware that if you have a parameterized statement or local variables, the optimizer can't use the filtered index.

1.11 If you made adjustments in step 10, re-run the query and record results

1.12 Finally, engineer out the stupid—that is, eliminate these frequently encountered inhibitors of performance whenever possible:

- Be aware that code-first generators (for example, EMF, LNQ, nHibernate) can bloat plan cache.

   Tip: Consider turning on OPTIMIZE FOR AD HOC WORKLOADS if you are using code-first generators.

- Look for abuse of wildcards (*), which can result in pulling back too many columns.
- Scalar functions and multi-statement functions get called for every row that gets returned, and can be abused.
- Nested views that go across linked servers can add processing time.
- Cursors and row-by-row processing can slow processing down.
- Join/query/index/table hints can significantly change how a query works. Use these only if you have exhausted all other possibilities.

SQL Server Performance Tuning
A 12 step program

Fellowmind