**SQL Server Performance Tuning and Monitoring Tutorial**

SQL Server is a great platform to get your database application up and running fast. The graphical interface of SQL Server Management Studio allows you to create tables, insert data, develop stored procedures, etc... in no time at all. Initially your application runs great in your production, test and development environments, but as use of the application increases and the size of your database increases you may start to notice some performance degradation or worse yet, user complaints.

This is where performance monitoring and tuning come into play. Usually the first signs of performance issues surface from user complaints. A screen that used to load immediately now takes several seconds. Or a report that used to take a few minutes to run now takes an hour. As I mentioned these issues usually arise from user complaints, but with a few steps and techniques you can monitor these issues and tune accordingly, so that your database applications are always running at peak performance.
In this tutorial we will cover some of the common issues with performance such as

- deadlocks
- blocking
- missing and unused indexes
- I/O bottlenecks
- poor query plans
- statistics
- wait stats
- fragmentation

We will look at basic techinques all DBAs and Developers should be aware of to make sure their database applications are performing at peak performance.

**SQL Server query tuning can be categorized into three broad steps:**

1. Basic query analysis
2. Advance query analysis
3. Facilitate tuning by using DB Performance monitoring tool

Here are 12 quick tips that can help a DBA improve query performance in a measurable way and at the same time provide certainty that the specific alteration has actually improved the speed of the query.

**1. Basic query analysis**

DBAs need visibility into all layers and information on expensive queries in order to isolate the root cause. Effective tuning requires knowing top SQL statements, top wait types, SQL plans, blocked queries, resource contention, and the effect of missing indexes. Start with the basics—knowing exactly what you're dealing with before you dive in can help.

```sql
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
```

**Tip 1: Know your tables and row counts**
First, make sure you are actually operating on a table, not view or table-valued function. Table-valued functions have their own performance implications. You can use SSMS to hover over query elements to examine these details. Check the row count by querying the DMVs.
**Tip 2: Examine the query filters, WHERE and JOIN clauses and note the filtered row count**
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.
**Tip 3: Know the selectivity of your tables**
Based upon the tables and the filters in the previous two tips , know how many rows you'll be working with, or the size of the actual, logical set. We recommend the use of SQL diagramming as a powerful tool in assessing queries and query selectivity.
**Tip 4: Analyze the additional query columns**
Examine closely the SELECT * or scalar functions to determine whether extra columns are involved. 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.
**2. Advanced query analysis**
**Tip 5: Knowing and using constraints can help**
Knowing and using constraints can be helpful as you start to tune. Review the existing keys, constraints, indexes to make sure you avoid duplication of effort or overlapping of indexes that already exist.To get information about your indexes, run the sp_helpindex stored procedure:

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

Results | Messages | Execution plan

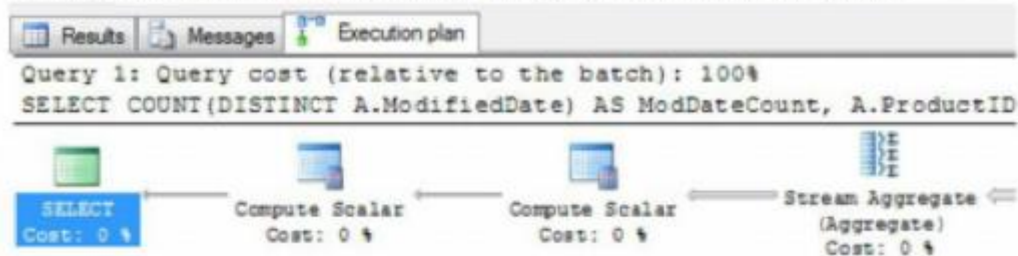
| | index_name | index_description | index_keys |
|---|---|---|---|
| 1 | AK_Sales_rowguid | nonclustered, unique, stats no recompute located o... | rowguid |
| 2 | cx_SalesSales | clustered, stats no recompute located on PRIMARY | SalesOrderID, SalesOrderDetailID |
| 3 | ix_modifiedSales | nonclustered, 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 |

**Tip 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.

```
SET STATISTICS IO ON
EXEC usp_Muerte

(265 row(s) affected)
Table 'Sales'. Scan count 33, logical reads 1529576, physical reads 0,
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, re
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, re
Warning: Null value is eliminated by an aggregate or other SET operati
```




**Tip 7: Record your results, focusing on the number of logical I/Os**

If you don't record the results, you won't be able to determine the true impact of your changes.

**Tip 8: Adjust the query based on your findings and make small, single changes at a time**

Making too many changes at one time can be ineffective as they can 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.

**Tip 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 tip 8 to examine other factors that may need adjusting. Keep making one change at a time, rerun the query and comparing results until you are satisfied that you have addressed all the expensive operations that you can.

**Tip 10: If 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. You can consider the existing indexes, a covering index and a filtered index for improvements.

**Tip 11: Rerun the query and record results**

If you have made adjustments, rerun the query and record those results again.

**Tip 12: Engineer out the stupid**

Lookout for frequently encountered inhibitors of performance like: code first generators, abuse of wildcards, scalar functions, Nested views, cursors and row by row processing.

**3. Use a DB Performance monitoring tool to facilitate query tuning.**

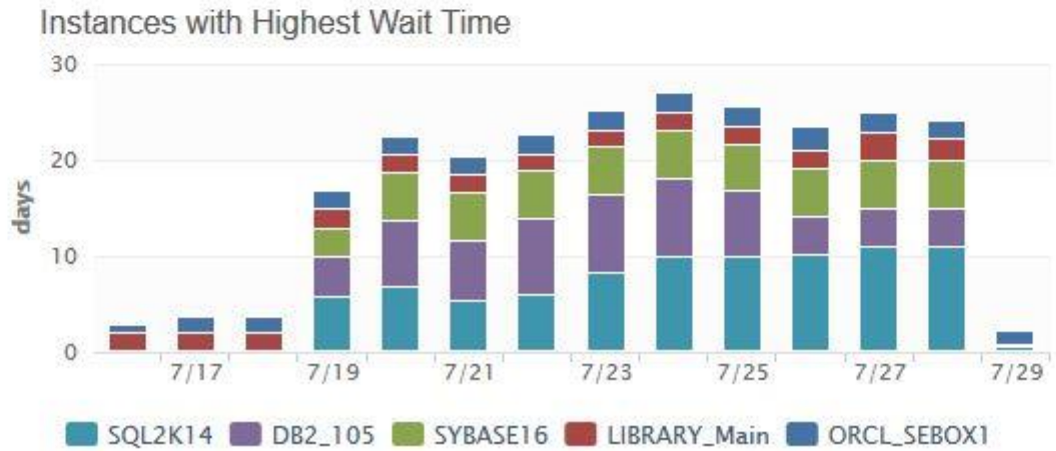
Traditional database monitoring tools focus on health metrics. Current application performance management tools provide hints, but do not help find the root cause.

solarwinds

Database Performance Analyzer

Performance

**Repository:** DATABASE PERFORMANCE ANALYZER (DPAv6-01)




| DB Instances | Wait Time | Query Advice | | CPU | |
|---|---|---|---|---|---|
| 21 / 21 MONITORING | 0 TRACKING HIGH | 1 CRITICAL | 0 WARNING | 2 CRITICAL | 2 WARNIN |

Type DB instance name | Apply Filter

| Database Instance ▲ |
|---|
| ⊞ DB2 |
| ⊞ MSSQLServer |
| ⊞ MySQL |
| ⊞ Oracle |
| ⊞ Oracle Library (CDB) |
| ⊞ Sybase |