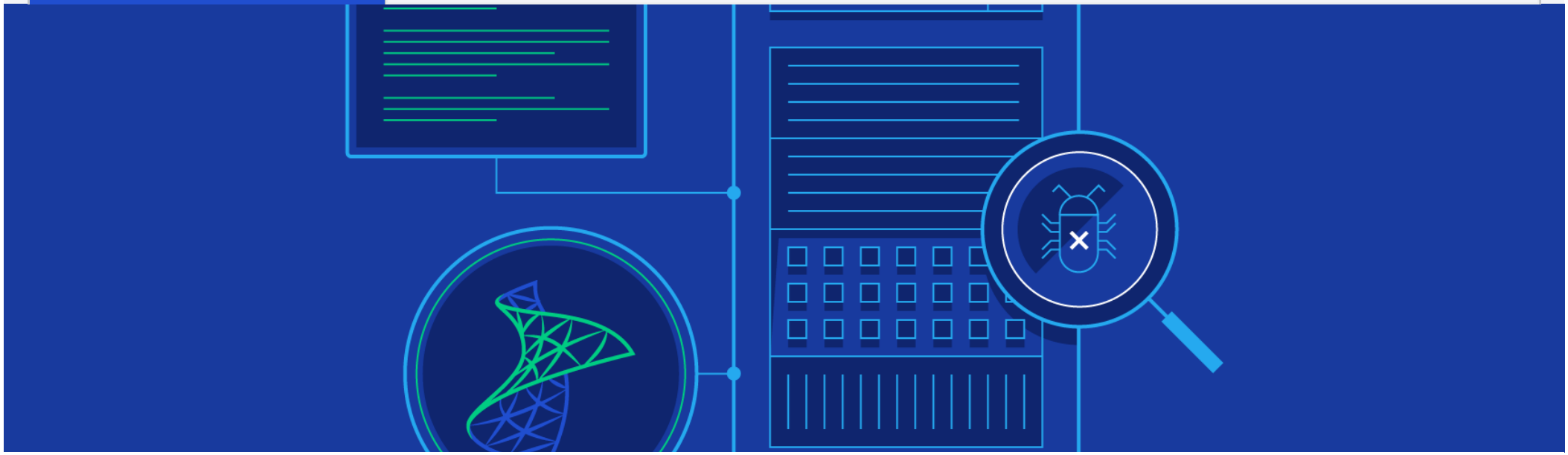

TECHNOLOGY 10 MINUTE READ

How to Tune Microsoft SQL Server for Performance

Sripal is a perfectionist website performance optimization engineer, experienced in front-end, back-end, as well as database development.

To retain its users, any application or website must run fast. For mission critical environments, a couple of milliseconds delay in getting information might create big problems. As database sizes grow day by day, we need to fetch data as fast as possible, and write the data back into the database as fast as possible. To make sure all operations are executing smoothly, we have to tune our database server for performance.

In this article I will describe a step-by-step procedure for basic performance tuning on one of the top database servers in the market: Microsoft SQL Server (SQL Server, for short).

#1 Finding The Culprits

As with any other software, we need to understand that SQL Server is a complex computer program. If we have a problem with it, we need to discover why it is not running as we expect.

From SQL Server we need to pull and push data as fast and as accurately as possible. If there are issues, a couple of basic reasons, and the first two things to check, are:

- The hardware and installation settings, which may need correcting since SQL Server needs are specific
- If we have provided the correct T-SQL code for SQL Server to implement

Even though SQL Server is proprietary software, Microsoft has provided a lot of ways to understand it and use it efficiently.

If the hardware is OK and the installation has been done properly, but the SQL Server is still running slowly, then first we need to find out if there are any software related errors. To check what is happening, we need to observe how different threads are performing. This is achieved by calculating wait statistics of different threads. SQL server uses threads for every user request, and the thread is nothing but another program inside our complex program called SQL Server. It is important to note that this thread is not an operating system thread on which SQL server is installed; it is related to the SQLOS thread, which is a pseudo operating system for the SQL Server.

Wait statistics can be calculated using `sys.dm_os_wait_stats` Dynamic Management View (DMV), which gives additional information about its current state. There are many scripts online to query this view, but my favorite is Paul Randal's script because it is easy to understand and has all the important parameters to observe wait statistics:

```
([wait_time_ms] - [signal_wait_time_ms]) / 1000.0 AS [ResourceS],
[signal_wait_time_ms] / 1000.0 AS [SignalS],
[waiting_tasks_count] AS [WaitCount],
100.0 * [wait_time_ms] / SUM ([wait_time_ms]) OVER() AS [Percentage],
ROW_NUMBER() OVER(ORDER BY [wait_time_ms] DESC) AS [RowNum]
FROM sys.dm_os_wait_stats
WHERE [wait_type] NOT IN (
N'BROKER_EVENTHANDLER', N'BROKER_RECEIVE_WAITFOR',
N'BROKER_TASK_STOP', N'BROKER_TO_FLUSH',
N'BROKER_TRANSMITTER', N'CHECKPOINT_QUEUE',
N'CHKPT', N'CLR_AUTO_EVENT',
N'CLR_MANUAL_EVENT', N'CLR_SEMAPHORE',
N'DBMIRROR_DBM_EVENT', N'DBMIRROR_EVENTS_QUEUE',
N'DBMIRROR_WORKER_QUEUE', N'DBMIRRORING_CMD',
N'DIRTY_PAGE_POLL', N'DISPATCHER_QUEUE_SEMAPHORE',
N'EXECSYNC', N'FSAGENT',
N'FT_IFTS_SCHEDULER_IDLE_WAIT', N'FT_IFTSHC_MUTEX',
N'HADR_CLUSAPI_CALL', N'HADR_FILESTREAM_IOMGR_IOCOMPLETION',
N'HADR_LOGCAPTURE_WAIT', N'HADR_NOTIFICATION_DEQUEUE',
N'HADR_TIMER_TASK', N'HADR_WORK_QUEUE',
N'KSOURCE_WAKEUP', N'LAZYWRITER_SLEEP',
N'LOGMGR_QUEUE', N'ONDEMAND_TASK_QUEUE',
N'PWAIT_ALL_COMPONENTS_INITIALIZED',
N'QDS_PERSIST_TASK_MAIN_LOOP_SLEEP',
N'QDS_CLEANUP_STALE_QUERIES_TASK_MAIN_LOOP_SLEEP',
N'REQUEST_FOR_DEADLOCK_SEARCH', N'RESOURCE_QUEUE',
N'SERVER_IDLE_CHECK', N'SLEEP_BPOOL_FLUSH',
N'SLEEP_DBSTARTUP', N'SLEEP_DCOMSTARTUP',
N'SLEEP_MASTERDBREADY', N'SLEEP_MASTERMDREADY',
N'SLEEP_MASTERUPGRADED', N'SLEEP_MSDBSTARTUP',
N'SLEEP_SYSTEMTASK', N'SLEEP_TASK',
N'SLEEP_TEMPDBSTARTUP', N'SNI_HTTP_ACCEPT',
```

```
N'SP_SERVER_DIAGNOSTICS_SLEEP', N'SQLTRACE_BUFFER_FLUSH',
N'SQLTRACE_INCREMENTAL_FLUSH_SLEEP',
N'SQLTRACE_WAIT_ENTRIES', N'WAIT_FOR_RESULTS',
N'WAITFOR', N'WAITFOR_TASKSHUTDOWN',
N'WAIT_XTP_HOST_WAIT', N'WAIT_XTP_OFFLINE_CKPT_NEW_LOG',
N'WAIT_XTP_CKPT_CLOSE', N'XE_DISPATCHER_JOIN',
N'XE_DISPATCHER_WAIT', N'XE_TIMER_EVENT')
AND [waiting_tasks_count] > 0
)
SELECT
MAX ([W1].[wait_type]) AS [WaitType],
CAST (MAX ([W1].[WaitS]) AS DECIMAL (16,2)) AS [Wait_S],
CAST (MAX ([W1].[ResourceS]) AS DECIMAL (16,2)) AS [Resource_S],
CAST (MAX ([W1].[SignalS]) AS DECIMAL (16,2)) AS [Signal_S],
MAX ([W1].[WaitCount]) AS [WaitCount],
CAST (MAX ([W1].[Percentage]) AS DECIMAL (5,2)) AS [Percentage],
CAST ((MAX ([W1].[WaitS]) / MAX ([W1].[WaitCount])) AS DECIMAL (16,4)) AS [AvgWait_S],
CAST ((MAX ([W1].[ResourceS]) / MAX ([W1].[WaitCount])) AS DECIMAL (16,4)) AS [AvgRes_S],
CAST ((MAX ([W1].[SignalS]) / MAX ([W1].[WaitCount])) AS DECIMAL (16,4)) AS [AvgSig_S]
FROM [Waits] AS [W1]
INNER JOIN [Waits] AS [W2]
ON [W2].[RowNum] <= [W1].[RowNum]
GROUP BY [W1].[RowNum]
HAVING SUM ([W2].[Percentage]) - MAX ([W1].[Percentage]) < 95; -- percentage threshold
GO
```

When we execute this script, we need to concentrate on the top rows of the result because they are set first and represent the maximum wait type.

We need to understand wait types so we can make the correct decisions. To learn about different wait types, we can go to the excellent Microsoft documentation.

Let's take an example where we have too much `PAGEIOLATCH_XX`. This means a thread is waiting for data page reads from the disk into the buffer, which is nothing but a memory block. We must be sure we understand what's going on. This does not necessarily mean a poor I/O subsystem or not enough memory, and increasing the I/O subsystem and memory will solve the problem, but only temporarily. To find a permanent solution we need to see why so much data is being read from the disk: What types of SQL commands are causing this? Are we reading too much data instead of reading less data by using filters, such as `where` clauses? Are too many data reads happening because of table scans or index scans? Can we convert them to index seeks by implementing or modifying existing indexes? Are we writing SQL queries that are misunderstood by SQL Optimizer (another program inside our SQL server program)?

We need to think from different angles and use different test cases to come up with solutions. Each of the above wait type needs a different solution. A database administrator needs to research them thoroughly before taking any action. But most of the time, finding problematic T-SQL queries and tuning them will solve 60 to 70 percent of the problems.

#2 Finding Problematic Queries

As mentioned above, first thing we can do is to search problematic queries. The following T-SQL code will find the 20 worst performing queries:

```
SELECT TOP 20
total_worker_time/execution_count AS Avg_CPU_Time
,Execution_count
,total_elapsed_time/execution_count as AVG_Run_Time
,total_elapsed_time
,(SELECT
SUBSTRING(text,statement_start_offset/2+1,statement_end_offset
) FROM sys.dm_exec_sql_text(sql_handle)
) AS Query_Text
FROM sys.dm_exec_query_stats
ORDER BY Avg_CPU_Time DESC
```

We need to be careful with the results; even though a query can have a maximum average run time, if it runs only once, the total effect on the server is low compared to a query which has a medium average run time and runs lots of times in a day.

#3 Fine Tuning Queries

The fine-tuning of a T-SQL query is an important concept. The fundamental thing to understand is how well we can write T-SQL queries and implement indexes, so that the SQL optimizer can find an optimized plan to do what we wanted it to do. With every new release of SQL Server, we get a more sophisticated optimizer that will cover our mistakes in writing not optimized SQL queries, and will also fix any bugs related to the previous optimizer. But, no matter how intelligent the optimizer may be, if we can't tell it what we want (by writing a proper T-SQL queries), the SQL optimizer won't do be able to do its job.

SQL Server uses advanced search and sorting algorithms. If we are good at search and sorting algorithms, then most of the time we can guess why SQL Server is taking particular action. The best book for learning more and understanding such algorithms is

The Art of Computer Programming by Donald Knuth.

When we examine queries that need to be fine-tuned, we need to use the execution plan of those queries so that we can find out how SQL server is interpreting them.

I can't cover all the aspects of the execution plan here, but on a basic level I can explain the things we need to consider.

- First we need to find out which operators take most of the query cost.
- If the operator is taking a lot of cost, we need to learn the reason why. Most of the time, scans will take up more cost than seeks. We need to examine why a particular scan (table scan or index scan) is happening instead of an index seek. We can solve this problem by implementing proper indexes on table columns, but as with any complex program, there is no fixed solution. For example, if the table is small then scans are faster than seeks.
- There are approximately 78 operators, which represent the various actions and decisions of the SQL Server execution plan. We need to study them in-depth by consulting the Microsoft documentation, so that we can understand them better and take proper action.

#4 Execution Plan Re-use

Even if we implement proper indexes on tables and write good T-SQL code, if the execution plan is not reused, we will have performance issues. After fine-tuning the queries, we need to make sure that the execution plan may be re-used when necessary. Most of the CPU time will be spent on calculating execution plan that can be eliminated, if we re-use the plan.

We can use the query below to find out how many times execution plan is re-used, where `usecounts` represents how many times the plan is re-used:

```
SELECT [ecp].[refcounts]
, [ecp].[usecounts]
, [ecp].[objtype]
, DB_NAME([est].[dbid]) AS [db_name]
, [est].[objectid]
, [est].[text] as [query_ext]
, [eqp].[query_plan]
FROM sys.dm_exec_cached_plans ecp
CROSS APPLY sys.dm_exec_sql_text ( ecp.plan_handle ) est
CROSS APPLY sys.dm_exec_query_plan ( ecp.plan_handle ) eqp
```

The best way to re-use the execution plan is by implementing parameterized stored procedures. When we are not in a position to implement stored procedures, we can use `sp_executesql`, which can be used instead to execute T-SQL statements when the only change to the SQL statements are parameter values. SQL Server most likely will reuse the execution plan that it generated in the first execution.

Again, as with any complex computer program, there is no fixed solution. Sometimes it is better to compile the plan again.

Let's examine following two example queries:

- `select name from table where name = 'pal';`

Let us assume we have a non-clustered index on the `name` column and half of the table has value `sri` and few rows have `pal` in the `name` column. For the first query, SQL Server will use the table scan because half of the table has the same values. But for the second query, it is better to use the index scan because only few rows have `pal` value.

Even though queries are similar, the same execution plan may not be good solution. Most of the time it will be a different case, so we need to carefully analyze everything before we decide. If we don't want to re-use the execution plan, we can always use the "recompile" option in stored procedures.

Keep in mind that even after using stored procedures or `sp_executesql`, there are times when the execution plan won't be re-used. They are:

- When indexes used by the query change or are dropped
- When the statistics, structure or schema of a table used by the query changes
- When we use the "recompile" option
- When there are a large number of insertions, updates or deletes
- When we mix DDL and DML within a single query

#5 Removing Unnecessary Indexes

After fine-tuning the queries, we need to check how the indexes are used. Index maintenance requires lots of CPU and I/O. Every time we insert data into a database, SQL Server also needs to update the indexes, so it is better to remove them if they are not used.

SQL server provides us `dm_db_index_usage_stats` DMV to find index statistics. When we run the T-SQL code below, we get usage statistics for different indexes. If we find indexes that are not used at all, or used rarely, we can drop them to gain performance.

```
SELECT
OBJECT_NAME(IUS.[OBJECT_ID]) AS [OBJECT NAME],
DB_NAME(IUS.database_id) AS [DATABASE NAME],
I.[NAME] AS [INDEX NAME],
USER_SEEKS,
USER_SCANS,
USER_LOOKUPS,
USER_UPDATES
FROM SYS.DM_DB_INDEX_USAGE_STATS AS IUS
INNER JOIN SYS.INDEXES AS I
ON I.[OBJECT_ID] = IUS.[OBJECT_ID]
AND I.INDEX_ID = IUS.INDEX_ID
```

#6 SQL Server Installation And Database Setup

When setting up a database, we need to keep data and log files separately. The main reason for this is that writing and accessing data files is not sequential, whereas writing and accessing log files is sequential. If we put them on the same drive we can't use them in an optimized way.

When we purchase Storage Area Network (SAN), a vendor may give us some recommendations on how to setup it up, but this information is not always helpful. We need to have a detailed discussion with our hardware and networking guys on how to keep data and log files separately and in an optimized way.

#7 Don’t Overload SQL Server

The primary task of any database administrator is to make sure the production server runs smoothly and serves customers as well as possible. To make this happen we need to maintain separate databases (if possible, on separate machines) for the following environments:

- Production
- Development
- Testing
- Analytical

For a production database we need a database with full recovery mode, and for other databases, a simple recovery mode is enough.

Testing on a production database will put lots of load on the transaction log, indexes, CPU and I/O. That’s why we need to use separate databases for production, development, testing and analyzing. If possible, use separate machines for each database, because it will decrease the load on the CPU and I/O.

#8 Transaction log, tempdb and memory

Log file needs to have enough free space for normal operations because an autogrow operation on a log file is time-consuming and could force other operations to wait until it is completed. To find out the log file size for each database and how much it is used, we can use `DBCC SQLPERF(logspace)`.

The best way to set up tempdb is to put it on separate disk. We need to keep the initial size as big as we can afford because when it reaches an autogrow situation, performance will decrease.

As mentioned before, we need to make sure that SQL server runs on a separate machine, preferably one without any other application on it. We need to keep some memory for the operating system, plus some more if it is part of a cluster, so in most cases around 2GB should do.

For mission critical environments, a millisecond delay in getting information can be a deal breaker.

Conclusion:

The procedures and suggestions discussed here are for basic performance tuning only. If we follow these steps, we may, on average, get around 40 to 50 percent improvement in performance. To do advanced SQL Server performance tuning, we would need to dig much deeper into each of the steps covered here.

TAGS BackEnd Databases SQL SQLServer

Sripal Reddy Vindyala
Freelance Software Engineer

ABOUT THE AUTHOR

Sripal is a full-stack developer with extensive experience in front-end, back-end, and database development and improvement. He is a talented individual with an eye for perfection and a great track record.

Hire Sripal

TRENDING ARTICLES

ENGINEERING › TECHNOLOGY

Build with Confidence: A Guide to JUnit Tests

ENGINEERING › BACK-END

Creating Your First GraphQL API

ENGINEERING › WEB FRONT-END

The Dart Language: When Java and C# Aren't Sharp Enough

ENGINEERING › MOBILE

How to Create a Swipeable UITabBar From the Ground Up

SEE OUR RELATED TALENTS

SQL Server | SQL | Database | Back-end

Toptal Developers

- Android Developers
- AngularJS Developers
- Back-End Developers
- C++ Developers
- Data Analysts
- Data Engineers
- Data Scientists
- DevOps Engineers
- Ember.js Developers
- Freelance Developers
- Front-End Developers
- Full-Stack Developers
- HTML5 Developers
- iOS Developers
- Java Developers
- JavaScript Developers
- Machine Learning Engineers
- Magento Developers
- Mixed Reality Developers
- Mobile App Developers
- .NET Developers
- Node.js Developers
- PHP Developers
- Python Developers
- React.js Developers
- Ruby Developers
- Ruby on Rails Developers
- Salesforce Developers
- Scala Developers
- Software Architects
- Software Developers
- Unity or Unity3D Developers
- Virtual Reality Developers
- Web Developers

WordPress Developers

View more

Join the Toptal® community.

Hire a Developer OR Apply as a Developer

MOST IN-DEMAND TALENT

iOS Developers

Front-End Developers

UX Designers

UI Designers

Financial Modeling Consultants

Interim CFOs

ABOUT

Top 3%

Clients

Freelance Developers

Freelance Designers

Freelance Finance Experts

Freelance Project managers

Specialized Services

About Us

CONTACT

Contact Us

Press Center

Careers

FAQ