SQL SERVER PERFORMANCE TUNING CHECKLIST

By Eric Blinn, MSSQLTips.com
March 2021

Performance Tuning Checklist

This checklist will serve as a starting point to help determine what steps might make sense as part of a SQL Server performance improvement effort. It provides a small amount of information for each checklist item and refers to other articles for more details when applicable.

The Checklist

There are 2 main sections to the checklist. The first section covers actions that might make sense when the entire SQL Server instance is running slow. The second section focuses on areas where specific queries or procedures are running slowly. A performance tuning effort often includes both parts.

Some of the items included in the checklist are:

- CPU
- Memory
- Disks
- Indexing
- Maintenance
- Query Plans
- Query Optimization

Server Level Items

Checklist Item	Description
☐ CPU Pressure	SQL Server instances need access to many CPU cycles to optimize query plans and execute queries. If the CPU is overloaded that can cause system-wide performance issues. There are many ways to measure CPU utilization. It can be watched live on a Windows machine using Performance Monitor. Also, by default SQL Server keeps the last 4 hours and 16 minutes of CPU history in the ring buffer which can be read on demand. This is a nice feature because it comes with a small amount of history. This query will show the output from the ring buffer in SSMS or ADS. This tip shows another way to measure CPU pressure on a SQL Server instance using wait stats. If CPU utilization is high consider tuning queries that use large amounts of CPU (see below in the checklist for details on how to accomplish this), moving non-SQL Server workloads to another machine, or adding CPU cores to the instance.
☐ Memory Pressure	A SQL Server workload is very data intensive. SQL Server prefers to do most of its reading from memory as RAM is generally much faster than disk. A SQL Server instance without enough memory will rely more on the disk subsystem and put increased pressure on it. SQL Server workloads can vary greatly so there isn't one best practice that can be applied to determine if there is memory pressure. These 2 tips [tip 1] [tip 2] can help measure memory statistics and the next checklist item on disk pressure will also be useful since a lack of memory will put so much extra pressure on the disk. If a SQL Server instance would benefit from more memory then one obvious solution is to add memory. Sometimes that isn't possible due to cost or licensing constraints. In those situations, consider compression to reduce the size of the data. Another option is to add IOPs to the disk subsystem so that it can better handle the extra work it is being asked to do.
☐ Disk Pressure	As stated in the previous checklist item, SQL Server is a very data intensive program. All write operations are sent to disk. SQL Server attempts to do reads via memory, but any reads that cannot be completed using memory are sent to disk. This means that memory pressure can become disk pressure. Check out this tip and this tip to learn more about measuring IO bottlenecks. There are a few things that can be done when a system is under disk pressure. The first is to attempt to reduce the overall need for IOPs by tuning large queries or moving some workloads, like reporting, to another instance. Another option is to add more memory in an attempt to reduce the IOPs requested of disk. The final option is to add IOPs to the disk subsystem -- perhaps by moving to a more modern, all-flash storage system.

Server Level Items

Checklist Item	Description
☐ Network Pressure	SQL Server often sends significant amounts of data through a network to client machines. Sometimes the network cannot keep up with the amount of data SQL Server is attempting to send which can lead to an overall sluggish system. A high level of ASYNC_NETWORK_IO waits would indicate such an issue. When this happens check to see if the network is saturated or if the client machines themselves are struggling to accept the data being sent by SQL Server. There is rarely a SQL Server solution to this wait type. This tip provides information to help figure out which wait stats are prevalent on the instance.
☐ SQL Server Patches	SQL Server releases updates several times per year and they often include fixes to known performance issues. If your instance is experiencing problems, consider applying the latest service pack or cumulative update to see if it helps. A list of patches can be found here, https://sqlserverbuilds.blogspot.com/. Details about the expected release schedule – including the lack of Service packs starting with SQL Server 2017 -- can be found here.
☐ Index Maintenance	Indexes are the lifeblood of SQL Server operations and their proper maintenance is an integral part any highly performing instance. Indexes become fragmented over time and this maintenance, reorganizing and/or rebuilding, is what defragments them so that they can operate as intended. This tip will show how to make an appropriate index maintenance plan.
☐ Statistics Maintenance	Every time a query is executed on a SQL Server instance, a program called the Optimizer reviews metadata about the data in the table(s) being queried in an effort to build an optimal query plan. It is important for this metadata to be reasonably accurate and up-to-date. SQL Server stores this metadata, which it calls “statistics”, on every index and many columns in the database. Learn more about these statistics and how to keep them up to date here.

Server Level Items

Checklist Item	Description
☐ Index Mix	When examining a query request, the optimizer looks for an index, or perhaps several indexes, that it can use to run the query efficiently. When it finds the right index on a table, it uses it. When it does not find an index that would help the query, it must move on to another less optimal index or even a scan. SQL Server tracks the results of these decisions. It keeps a running list of indexes it wished existed during query optimization, but didn't. It also keeps track of how many times it has used the indexes that do exist. These 2 tips will help figure out which indexes are not being used and may be able to be dropped and which indexes should be created so that the optimizer can quit scanning those tables so often.

☐ Compatibility Level	Microsoft has improved the query optimizer over the years, but those improvements are only available if running on the latest compatibility level. Compatibility level is a setting on each individual database.
	
	This tip is a little older, but explains how a database might come to have an older compatibility level.
	
	The query below will show the compatibility level of each database on an instance.
	
	110 is SQL Server 2012 compatibility level
	120 is SQL Server 2014
	130 is SQL Server 2016
	140 is SQL Server 2017
	150 is SQL Server 2019
	
	`SELECT name, compatibility_level FROM sys.databases;`
	
	This is the code needed to change the compatibility level of a database called WideWorldImporters to SQL Server 2019 compatibility.
	
	`ALTER DATABASE [WideWorldImporters] SET COMPATIBILITY_LEVEL = 150;`

☐ Virtual Log Files	The transaction log of every SQL Server database is made up of many virtual log files (VLFs). A database that has too many VLFs can perform poorly. This page is a great description of VLFs. This tip explains how having too many VLFs can induce performance problems and how to fix such a problem.

Server Level Items

Checklist Item	Description
☐ Identify Slow Queries	Poorly performing queries, especially those that run often, can cause performance issues all over a SQL Server environment. Identifying them isn't always easy, but SQL Server offers some built-in tools to help find them so that they can be optimized. Use the information from this tip to help identify poor performing queries. There were some significant changes to the tools in SQL Server 2016 that are covered here. Check out the next section of the checklist for advice on how to improve the performance of any queries that turned up using this method.

Query Level Items

Checklist Item	Description
☐ Query missing an index	Most SQL Server queries should use indexes to quickly seek the necessary rows in each table being referenced in the from clause. When an ideal index isn't available, SQL Server will either need to use a sub-optimal index or scan the table which can be a much slower operation. This tip explains how to see if the optimizer thinks an index will make a query run faster and also offer an estimate of how much faster!
☐ Query doing large key lookup	Even if a query is able to find an optimal index on every table listed in a from clause that doesn't mean the performance tuning effort is complete! Often index seeks come with key lookups, an operation whereby SQL Server has to query the clustered index in addition to the nonclustered index used for the seek. While those operations aren't necessarily bad, if they are taking up the bulk of the query execution time it may make sense to try to get rid of them. Here are 2 tips that help further explain what a key lookup is, how to diagnose if they are significant, and how to get rid of them if desired. Tip 1. Tip 2.
☐ Query uses loop or cursor	Did a search for slow performing queries return a procedure that uses a cursor or a loop? That should not come as a surprise as these constructs are among the slowest ways to write a SQL query. Cursors are popular among newer developers because the logic works much in the way the human brain works. *I'll scroll through the rows. For each row, if it meets criterion A and criterion B then I'll update the status column.* Unfortunately, this is not how the SQL Server engine wants to work. When writing TSQL code don't think about what needs to be done to each row. Instead, think about what needs to be done to each column. Code written in this style is called set-based. *I'll update the status column for any row where criterion A and criterion B are true.* This page from a performance tutorial works through an example comparing set-based code to looping code that accomplishes the same thing.

Query Level Items

Checklist Item	Description
☐ Query compiles for a long time	Sometimes a query will run slowly, but it doesn't have any telltale signs of slow performance like a large scan or a key lookup. In a situation such as this it might not be the query that is taking all that time, but rather the optimizer is having a hard time coming up with a plan. Review this tip to learn how to differentiate between actual query execution time and time spent by the optimizer building a plan. If it turns out that the parse and compile time is the cause of slow query execution consider the physical length of the query as this is the most likely reason for the long compile time. Does the query rely on a view? If so, then consider the size of the view as well. Views that are based on views can expand very quickly and become a compilation nightmare.

☐
Query doesn't use searchable arguments
	If a query isn't using an index and the optimizer doesn't request one there might be an issue with an argument in a join predicate or where clause. Only certain types of arguments are searchable, meaning that they CAN use an index.
	
	Consider this where clause. While there may be an index on the orderdate column, this argument is NOT searchable. That is because there is no index on the newly-calculated column that is the result of the DATEDIFF calculation. It will ALWAYS result in a scan.
	
	`WHERE DATEDIFF(dd, orderdate, GETDATE() < 5`
	
	Now consider this where clause. It accomplishes the same thing, but compares orderdate to the result of a DATEADD function that depends on no columns. That function can be calculated once and then the resulting value compared to the column orderdate. If orderdate is indexed then the index can be used!
	
	`WHERE orderdate >= DATEADD(dd, -5, GETDATE())`
	
	Whenever building join predicates, where clauses, or having clauses, be sure that the arguments are searchable.

Query Level Items

Checklist Item	Description
☐ Properly Using Dynamic SQL	Dynamic SQL is a very polarizing topic in the SQL Server universe. This author loves dynamic SQL so long as it is done properly. One of the most common issues with this feature is running it with values that appear to be hard coded to the optimizer. Consider this query. If it is generated with a series of different city names it will need to be compiled every time. While that isn't a big deal for this small query, it could easily become a problem if the query is longer and requires more compilation resources. This type of dynamic statement also opens the program up for SQL injection attacks if the city name is entered by a user. `exec sp_executeSQL N'SELECT CityName, CityID FROM Application.Cities WHERE CityName = ''Springfield'';';` This tip explains how to better use dynamic SQL and how to protect a program from SQL injection attacks by using parameters that are vetted.
☐ Query returning too many columns	A common performance problem occurs when SQL Server is asked to return more columns than are needed by an application. An example of this would be an application that is looking to display a customer name and phone number, but to save programming time the developer wrote a simple query to select all columns from the customer table that could be reused by many different customer-related operations. `SELECT * FROM dbo.Customer WHERE CustomerCode = 123;` This can lead to network latency, wasted memory on the server and client, excess key lookups, and inefficient index requests. While it takes more time to write several targeted queries and those queries aren't as reusable, it is generally worth that effort to get better performance from a SQL Server.
☐ Query returning too many rows	Another common performance problem is allowing an application to retrieve too many rows. Imagine an application that has a customer search feature. The user types in a single letter and hits search. The database may find and want to return 50,000 rows to the client. But it is wildly unlikely that the user is ever going to look at anything beyond the first few hundred rows. Meanwhile, SQL Server spent the memory, CPU, IO, and network resources needed to gather and send all 50,000 rows and the client had to receive and store them all in memory. One solution to this problem might be to refuse to perform a search until more limiting search criteria are provided. Another solution could be to limit the results to the first 1,000 rows no matter what is used as search criteria.

About Author

Eric Blinn has been a SQL Server DBA and Architect in the legal, software, transportation, and insurance industries for over 10 years. Currently he is the Sr Data Architect for Squire Patton Boggs, a leading provider of legal services with 47 offices in 20 countries. Eric is a 2018-2019 Idera Ace and has co-authored 2 Idera Whitepapers. He has been a presenter at PASS Summit, IT/DevConnections, SQLSaturdays, the in.sight transportation conference, MSSQLTips.com and the Ohio North SQL Server User's Group.

About MSSQLTips

MSSQLTips.com produces and provides technical content related to SQL Server. MSSQLTips was started in 2006 by Edgewood Solutions, LLC, SQL Server industry experts that saw the need for a different approach to learning about SQL Server. Our mission is to help educate and solve real world problems for the global SQL Server community every second of the day.