Throwing Hardware at SQL Server Performance problems?
Think again, there’s a better way!

Written By:
Jason Strate, Pragmatic Works
Roger Wolter, Pragmatic Works
Bradley Ball, Pragmatic Works
As DBAs, we continually face the challenge of improving the performance of our databases. Even if we do make our users satisfied, it doesn’t always last very long. This paper discusses some of the more common ways to identify and improve performance problems without additional hardware.

Typically there are three basic approaches to improving database performance:

- Buy bigger, faster hardware
- Take advantage of new database features that offer improved performance
- Make code and configuration changes to make better use of available resources

For many DBAs the first approach seems the most attractive. Manufacturers are coming out with newer processors with many more cores, more dense memory, and faster disks all the time, so why not go for the quick fix and replace the hardware? In some cases this may be the right move, but what happens when you talk your management into spending more than you make in a decade on new hardware and the system is still too slow? If the issue wasn’t hardware performance, or if you talked them into a new server and the issue was disk performance, you’re going to be cleaning egg off your face for quite a while.

This paper talks about how to monitor your database performance so you can be sure you’re fixing the right problem and maybe even finding and fixing performance problems before they become an issue. SQL Server 2014 has quite a few features to improve performance so choice two might be a viable option for many applications, but this option may not work for everyone, so understanding what is causing your performance issues is the key to any solution.

To understand the true cause of performance issues, you must be able to quickly identify trends in performance issues before your users do. I often get questions like, “The Seconds per Read counter is 40ms. Do you think that’s what’s causing the problem?” I have to say, “Maybe. What was it last month when you weren’t having a problem?” They almost never know. Very few SQL Server performance statistics will definitively identify a problem unless you can associate the start of a performance issue with a change in a performance statistic. Creating and maintaining a baseline of your current performance numbers is key to identifying changes in performance when they happen.

While SQL Server provides many ways to gather performance information – Traces, Extended Events, Dynamic Management Views, Perfmon, etc. – most DBAs don’t have the time to become expert at gathering and interpreting the information these provide. The most practical way to gather and analyze this performance data is to use a third party tool such as SolarWinds Database Performance Analyzer (DPA) that does most of the heavy lifting of analysis for you.

For some of the more common performance problems, you might consider reviewing SQL Server configuration settings and making software changes before investing in a new hardware infrastructure. Even if you currently aren’t experiencing performance problems, you should start gathering performance statistics so that when you do run into a problem, you can quickly identify what changed to cause the problem.
**OPTIMIZE THE HARDWARE YOU HAVE**

When you have a performance issue, you should be sure that you’re getting the most out of the hardware you have now before you ever consider new hardware. Specifically, you should check the Power Saver settings and the HBA Queue depth.

---

**POWER SAVER SETTINGS**

Microsoft® Windows® servers can slow down during off-peak usage to save power and run cooler. While this is definitely a good feature, it can cause SQL Server performance problems. When the server isn’t being pushed hard, the processors slow down. This means that query response times go down when the server isn’t working hard. The easiest way to fix this is to change the Power Saver setting from the default of “Balanced” to “High Power”. This will make query response times consistent, but will use more power. If you want better response time while still saving power, you can try the process explained here: [https://support.microsoft.com/kb/2207548?wa=wsignin1.0](https://support.microsoft.com/kb/2207548?wa=wsignin1.0)

I generally recommend using “High Performance”.

I once worked with a customer who had just tripled the number of processor cores and memory they had powering the SQL Server database. Needless to say, they were rather disappointed when their new processors showed 70 – 89% usage when running under load. When we changed their power-saver setting to high performance, processor usage decreased to under 20%.

---

**HBA QUEUE DEPTH**

SQL Server storage IO tends to happen in bursts which can temporarily exceed the SAN’s ability to process them. To handle bursts in IO, Host Bus Adapters (HBA) have a queue to store IO commands until they can be sent to the SAN controller. Most HBAs default to a queue depth of 16 or 32. This is usually too small for SQL Server. The HBA queue depth should be 64 or 128 for SQL Server. I recommend starting with 64, and if that improves performance, then try 128. If the OS Disk Queue Length goes into double digits frequently or if it has been increasing over time, the HBA queue depth is a likely candidate.

Last year I worked on a Service Broker application that would occasionally get very far behind in message processing several times a week. After analyzing the application we determined there were no quick fixes and the application needed some significant re-architecture. While we were analyzing the performance we noticed that the HBA queue depth was set to 16. This was easy to fix so we asked the storage team to increase this to 64. Rewriting the application to a couple months, but the HBA queue depth change improved disk performance enough that the problem never happened during the whole time the application was being rewritten. The customer was satisfied that the rewrite was the right long-term solution, but the short-term HBA solved the problem with a five-minute effort.
OPTIMIZE STORAGE USAGE

Most DBAs have heard that putting logs, data, and tempdb on separate volumes is a best practice. This is very important for performance so make sure you do it. If you have some volumes that are faster than others – RAID 10 vs RAID 5, or SSD vs spinning media, the log and tempdb will benefit most from the added speed. Tempdb IO speed is most important if you use one of the snapshot isolation options or if your application creates a lot of temp tables.

If you already have your storage split up nicely, the next important way to reduce storage IO demand is by using data compression. Very few databases using modern processors are processor bound, but many of them are IO bound so it’s rare for compression to not improve performance significantly. That being said, if your processor usage is consistently over 70%, you need to address that before using compression. Compression not only saves storage space, it also increases the amount of data contained in each IO operation so the number of IOs required is reduced. The one issue I have run into is that compressed indexes take longer to rebuild so you may have to take that into account when planning your compression and maintenance strategy.

The last method to reduce disk IO is to change the application to use less IO. One of the biggest contributors to disk IO is table and index scans. See the section in this paper on reducing the number of scan operations for more information. Be aware that what looks like a disk IO problem might be caused by too many disk scans in the application.

DPA monitors many disk statistics that can be very helpful in assessing storage issues. The metrics I like to watch are seconds/ read and seconds/write. Ideally for a well-tuned SAN, seconds/ read should be under .010 (10ms) and seconds/write should be under .005 (5ms), but you may get adequate performance from a storage subsystem with numbers considerably higher than this. As with most metrics, the important thing to look at is how the current numbers compare to a time when performance was adequate. The history maintained by DPA should give you that information. DPA also displays nice graphs of trends that should help you find worsening disk performance before your users notice a problem.
PARALLELISM

If you have been paying attention to hardware trends you have probably noticed that processor clock speeds increased from a few megahertz to about 3 gigahertz - a thousand fold increase in just a few years - but then for the last decade, clock rates have been pretty constant. New generations of microprocessor technology improve performance by increasing cache sizes, adding caches and increasing the number of cores per chip. This is mainly due to approaching some limits in the current chip technology. One of the effects of this change is that applications have to get better at running many tasks in parallel to take advantage of the number of cores available.

SQL Server has two ways to take advantage of parallel processors: running multiple queries simultaneously and running a single query on multiple processors. Running multiple queries in parallel is controlled by the SQL Server schedulers and happens automatically. If your server doesn’t have hyper-threading enabled, you might want to turn it on to increase parallelism. When hyper-threading was first introduced, it sometimes decreased performance, but it has been years since I have seen issues with it.

Using parallel processors to execute a single query is a bit more complex and requires some tuning to work well. There is a tradeoff between giving enough parallel resources to a query so that it runs quickly and dedicating too many resources to a single query so that it impacts the performance of the rest of the system.

The instance parameter MAXDOP controls how many parallel processors a query can use. The default value for MAXDOP is 0 which means the optimizer can chose to use all the processors in the system to execute a single query. All the processors is generally too much to dedicate to a query, unless you are running one or two huge queries at a time as part of a nightly processing job. The right value for MAXDOP depends on what kind of queries you run: higher for OLAP systems with a few very big queries and lower for OLTP systems. I generally start with setting it to the number of processors in a NUMA node, usually 6 or 8. If your system doesn’t have NUMA nodes, then 8 is a good place to start.

If DPA shows consistently high values for the CXPACKET wait type, it means your server is doing a lot of parallel queries. When the SQL Server engine splits a query into multiple execution threads, it has to wait for all the threads to complete before doing single threaded operations. The time spent waiting is recorded as a CXPACKET wait. In a data warehouse that does primarily large queries on large tables, high CXPACKET wait times may be normal. But in an OLTP system, this may be a sign of too many parallel queries. Parallelism is relatively expensive so it should be reserved for the very large queries that need it. The SQL Server optimizer always computes a serial plan first, and then if the cost of the serial plan is large enough, it computes a parallel plan and compares the serial and parallel costs to determine which one to use. The cost that must be exceeded before a parallel plan is computed is the “Cost Threshold for Parallelism” which is an advanced instance property. The Cost Threshold for Parallelism defaults to 5 which is much too small for a modern server. This should be set to at least 20 or 30.

I worked with a service bureau who had just recently signed their biggest customer and bought a 64 core server to ensure they could meet their customer’s demanding SLA. They were very disappointed to find that even though they had twice as many cores as they had for any of their other customers, the performance wasn’t significantly better. We found that changing the MAXDOP from 0 to 8 and the cost threshold from 5 to 30 more than doubled the performance.
MEMORY

Memory is generally the most critical factor in SQL Server performance. If a query can find the data it needs in memory, it may execute hundreds of times faster than if it has to go to disk. While adding memory will almost always help SQL Server performance, there are limits to how much memory a processor can use so optimizing the usage of memory is critical.

BUFFER CACHE

Optimizing storage performance through data compression also optimizes memory usage because the data pages are stored in memory in compressed format. If compression decreases disk space by 20% it will also increase the amount of data that can be cached in buffer memory by 20%.

You should frequently check your servers for signs of memory stress. One of the best ways to see memory stress is by monitoring the Page Life Expectancy (PLE). PLE is the average length of time that a page of data will stay in memory before it is thrown out because the buffer manager needs the space for new data. The longer a page stays in memory, the more likely it is to be there when the SQL engine needs it. The old rule of thumb was that as long as the PLE stayed above 300 (5 minutes) the memory was adequate. This really isn't true anymore. I have seen memory stress in systems where the PLE never went below 1000. The thing to look for is long, steep drops in the PLE. This is a sign that many pages are being thrown out of memory. You will see from the PLE graph that it can take many minutes for the PLE to return to normal after one of these drops. The gradual buildup of the PLE to normal levels represents thousands of IOs loading data into memory and because the data has to be read from disk, performance suffers. These sharp drops in PLE are almost always caused by table or index scans which read significant parts of a table into memory. Many scans can be eliminated by proper indexing and well-written queries. This will be discussed in the next section. Note that sharp PLE drops can also be caused by DBCC or index rebuilds which read large amounts of data. These drops can’t be avoided so it’s safe to ignore them.

Another sign of memory stress is that dirty pages (pages that contain data which has been changed but not yet persisted to disk) are written to disk so the memory can be freed. Dirty pages are normally written to disk during checkpoint operations that happen about once a minute, but if memory is needed between checkpoints, the lazy writer process writes dirty pages to disk. Monitoring the number of lazy writer pages is another way to detect memory pressure. A few lazy writes is normal, but if the lazy writer is writing many pages over an extended period, the system is suffering from memory pressure.
PLAN CACHE

Another important use for memory in SQL Server is the plan cache. Query plans are expensive and time consuming to compile, so SQL Server caches the plan it compiles and reuses it the next time it executes the same query. To speed up plan lookup, a hash is computed over the text of the query and compared to the hash for queries stored in the plan cache. This means that if the incoming query is not exactly the same character for character, including case and white space, the same as a query stored in the plan cache, a new plan will be compiled and stored in the plan cache.

One of the most common issues with the plan cache is caused by queries that look like this:

UPDATE ORDER_HEADER SET STATUS = 2 WHERE ORDER_NUMBER = 2837419

Because the ORDER_NUMBER is part of the query text, every time this query is run for a different order, a new plan is compiled and cached. In a very short time, the plan cache will be filled with tens of thousands of copies of this query – wasting memory and throwing valuable plans out of the cache to make room. The right way to fix this is to rewrite the query so the ORDER_NUMBER is a parameter and the plan can be reused. This isn’t always possible so SQL Server provides an OPTIMIZE FOR ADHOC QUERIES configuration parameter to address this issue. When this parameter is set, the plan is cached the second time the query is executed. This means that queries that are only executed once are not cached. I have never seen this parameter cause a problem so I recommend setting it in all cases. At one customer we found that out of about 30,000 plans in the cache, almost 29,000 were the same query with a different customer number. Setting the ADHOC QUERY option reduced the size of the plan cache by a factor of 8 and significant improve performance.
INDEXES AND QUERY TUNING

As mentioned previously, one of the significant contributors to excessive IO and memory usage is scans of tables and indexes. Scans read all rows in a table or a range of rows in a table. Reading a large number of rows can exhaust the free space in the buffer cache and force the cache to evict pages to make room for the pages required for the scan. This not only requires IOs to read in the rows being scanned, but the rows evicted from the cache to make room for the scanned rows must be read in again the next time they’re needed. Scans are not always bad. If not a lot of rows are required, a scan in often the best choice for performance. There are also some operations like SUM or COUNT which have to read all the rows to complete.

TABLE AND INDEX SCANS

Using DPA, you can find scans by looking at the plans for the most expensive queries tracked by. If the query plan contains scan operations, find out why the optimizer chose to do a scan and decide if the scan is a problem. There are several reasons why the optimizer will choose to scan an index or table:

1) The query needs to access all the rows. The query may not have a where clause or the where clause selects a large number of rows. These scans can’t be avoided.

2) The query accesses so few rows that a scan is the fastest way to execute the query. Look at the estimated rows returned and if this is a reasonably small number, ignore this scan. If the estimated number of rows is much lower than the actual number of rows scanned, the statistics for the table need updating.

3) There is no index on the columns being scanned. If there isn’t an index then a scan is the only option. Try putting an index on the column or columns that the scan is looking at.

4) There is an index, but the optimizer can’t use it. There are a couple common issues here:

   a. The query involves an implicit data type conversion and the optimizer can’t use the index because it has to convert all the values in the column to do the comparison. For example if the where clause is WHERE fname = @first_name and fname is varchar and @first_name is nvarchar then all the fname values must be converted to nvarchar because nvarchar has a higher precedence than varchar. Datatype precedence is documented here: http://msdn.microsoft.com/en-us/library/ms190309.aspx Implicit type conversion always converts the lower precedence type to the higher precedence type. The way to avoid the scan in this case would be to change @First_name to varchar so the index on fname can be used.

   b. The query uses a scalar function on the column used in a comparison. Here again, every row has to be read and have the function applied before the comparison can be done. A good example of this is WHERE UPPER(fname) = ‘JOE’

This is just the tip of the iceberg for indexing and query optimization. Many books have been written about it and I recommend you read one to get a more in-depth introduction to the subject.
TRANSACTION LOGS

One of the most critical determinants of database performance is transaction log write speed. Transactions can’t commit until the commit record is written to the transaction log. It’s pretty common to see applications with no BEGIN TRANSACTION … COMMIT statements. Some people think this means they aren’t using transactions. Relational databases always use transactions so if there are no transaction boundaries defined, each statement is its own transaction. This means that each statement has to wait for a transaction log write to complete. This puts a lot of stress on the transaction log which can adversely impact performance. Very long transactions are also a performance issue because locks are held until the transaction completes. Grouping statements into logically related transactions can improve performance by reducing the waits on transaction log writes.
**WAIT ANALYSIS**

So far we have covered several wait types and how they impact SQL Server performance. The next step is to determine which waits are involved in the queries that are causing issues. DPA includes a Response Time Analysis (RTA) feature that breaks the total response time into the wait states involved in producing the result. A careful analysis of the wait times can help determine which issues should be addressed to improve query performance. For example, if IO waits are a significant part of the query processing, you may want to consider adding an index to reduce the IO. If the biggest wait is ASYNC_NETWORK_IO, you will want to talk to the network team.

If you’re not sure which queries are causing issues, DPA will tell you which queries are taking the most time, which queries run most often, and which queries are using the most resources. Once you have found the heavy hitters, you can use RTA to find the significant waits and determine how to improve performance.

You can also analyze wait types at the instance level and correlate them with system resource metrics to make sure you are addressing the right problem. For example, if CXPACKET waits are high and Page Life Expectancy is low, you may be able to address this issue by changing instance-level parameters or buying more memory.

You should look for patterns in performance over time. There are normally peaks in resource usage at the same time of day or the same day of the week for example. DPA stores historical data in a repository so you can find performance trends like these more easily. In most cases you have little control over when these peaks happen, but it makes sense to schedule the things you do control – backups, maintenance, scheduled jobs, etc. – so that they happen at off-peak times. I once fixed a significant performance issue by staggering the start times of nightly processing jobs instead of starting them all at the same time.

I once worked on a file processing application that loaded ftp files from clients into a database for processing. As the number of files they loaded increased they reached a point where the SQL Server processors were maxed at 100% and processing was getting so far behind that they were receiving customer complaints. They ordered a significantly larger system, but while they were waiting for it to get delivered they asked me to try to improve performance of their existing system. We analyzed which queries were using the most resources and found to our surprise that a query that scanned the input tables for unprocessed data was using over half the total resources of the system. This was surprising because this query didn’t actually process the input. It just scanned the input for unprocessed data. After digging into this query we found it took about 30 seconds to run and it was scheduled to run every 5 seconds. Changing the schedule to run every 2 minutes instead of every 5 seconds dropped the processor usage from 100% to about 40% and the processing was current for the first time in several months.

You can also use DPA to track the performance of a query over time. This might tell you when it started performing poorly – possibly because the query plan changed, a lot of rows were added, or another application was deployed. When you’ve tuned or changed something, you can use DPA to determine whether performance improved, regressed or stayed the same.
SUMMARY

There are quite a few ways to improve SQL Server performance that don’t involve a significant investment in new hardware. Before you start bugging the boss for beefier servers or the latest Solid State Storage (SSDs), I recommend doing at least the following:

- Make sure you’re working on the right problem, review trends and analyze the issues
- Look for low hanging fruit
  - Check Power Saver and HBA Queue Depth Settings
  - Set MAXDOP to 8 and Cost Threshold for Parallelism to 30
- Implement data compression
- Set the OPTIMIZE FOR ADHOC QUERIES option
- Set up monitoring to check for
  - Table Scans
  - PLE
  - Lazy Writes
  - High Seconds Per Read and Seconds Per Write
- Use response time analysis and wait types to add indexes and change queries to fix the most expensive queries.

The key thing to remember is that while there are many things you can do to improve performance, you have little chance of fixing it, unless you first find out what the problem is. Gathering information, establishing baselines, and maintaining history are all critical to maintaining high performance. The fundamental performance analysis and resolution techniques covered in this paper apply whether you use a tool like DPA to gather, maintain, and analyze performance data or you do the analysis yourself with the SQL Server facilities.

Remember that while investing in the latest and greatest hardware toys is a tempting way to improve performance, throwing hardware at a problem without understanding what is causing the problem is a very risky approach. Use what you have learned to come up with an informed approach to finding and resolving performance problems.