5 REASONS YOUR SQL SERVER IS SLOW

Common Query Problems & How To Optimize or Fix Them

BY ROBERT L DAVIS

idera® | SQL Server Whitepaper

This whitepaper is a guide to the 5 most common things that cause your SQL Server to run more slowly than it should. These are issues that, once addressed, will allow your SQL Server instances to run faster. Best of all, very little work is required to fix them.

Everybody would love to find the mythical “go faster” switch in SQL Server. Unfortunately, this switch just doesn’t exist. There are, however, many things you can do (with very little effort) that can speed up your SQL Server instances—things that are needlessly slowing down your SQL Server and causing it to work harder.

These tips are what I like to call Quick Performance Wins, because they require a minimum amount of work to implement and can deliver large improvements in performance. These tips are things that both database administrators and SQL developers alike can use to get quick wins:

1. **BALANCED POWER PLAN**
2. **DEFAULT PARALLELISM SETTINGS**
3. **DATA TYPE MISMATCHES**
4. **FUNCTIONS ON COLUMNS**
5. **MAX SERVER MEMORY AT DEFAULT SETTING**

ABOUT THE AUTHOR

Robert L Davis is a senior database administrator and technical lead at Microsoft. He is a speaker and a trainer as well as a writer for SQL Server Magazine, and co-authored “Pro SQL Server 2008 Mirroring” (Apress).

Blogs: www.sqlsoldier.com **Twitter:** @sqlsoldier

1 BALANCED POWER PLAN

Windows Server 2008 introduced Power Plans. The default power plan is “balanced power plan”—which means that Windows may throttle the power to the CPUs to save energy. That sounds great in terms of green computing, an initiative that focuses on energy-efficient computing, but it is just not appropriate for SQL Server. This is particularly true with today’s per-core licensing of SQL Server. If you are willing to run SQL Server at 67% or 50% CPU power, you would be better off disabling one-third or half of the CPUs and saving hundreds of thousands of dollars in licensing.

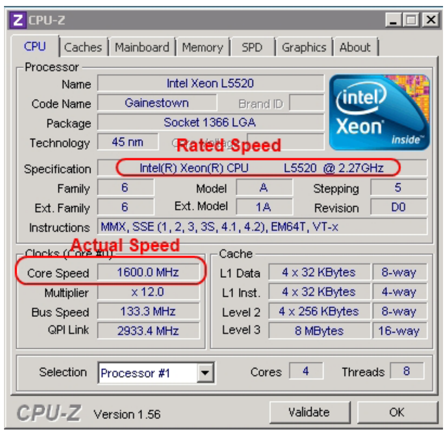
There is no legitimate reason why CPUs for production SQL Servers should be power-throttled.

The concept sounds good on paper: it throttles CPUs down when they are idle and back up when the server is busy. In reality, the CPU-throttling mechanism does not respond very well to CPU pressure. It requires a long, sustained period of very high CPU utilization to trigger a power increase in CPUs. In short, it’s not responsive to high CPU events. But that’s not even the worst thing about the CPU power being throttled.

When the CPU power is throttled down, everything runs slower on SQL Server. It does not even take high CPU utilization to feel the hit from power throttling. If the CPUs are throttled, everything takes longer to run. Even simple queries on a non-busy server can take up to twice as long to execute. SOS_Scheduler waits and lock waits may go up because contention on CPU time and resources are increased.

When every transaction takes longer, it can have a domino effect. The chance for a deadlock to occur is greater; concurrency is reduced because more queries end up running at the same time. The performance impact of this setting can manifest itself in many different ways.

There are two easy ways to see if the CPUs are getting power throttled. You may already be aware of the very popular free tool “CPU-Z,” downloadable from www.cpuid.com. This tool can be run as a stand-alone executable on a server, does not require installation, and is safe to use on a production server.



```
Processor
Name            Intel Xeon L5520
Code Name       Gainestown              Brand ID
Package         Socket 1366 LGA
Technology      45 nm
Specification   Intel(R) Xeon(R) CPU     L5520  @ 2.27GHz
Family          6        Model       A       Stepping   5
Ext. Family     6        Ext. Model  1A      Revision   D0
Instructions    MMX, SSE (1, 2, 3, 3S, 4.1, 4.2), EM64T, VT-x

Clocks (Core #0)                 Cache
Core Speed   1600.0 MHz          L1 Data   4 x 32 KBytes    8-way
Multiplier   x 12.0              L1 Inst.  4 x 32 KBytes    4-way
Bus Speed    133.3 MHz           Level 2   4 x 256 KBytes   8-way
QPI Link     2933.4 MHz          Level 3   8 MBytes         16-way

Selection   Processor #1      Cores  4     Threads  8

CPU-Z  Version 1.56
```

The image shows the output from a real production server I investigated for CPU pressure several years ago. The “Specification” on the output is the speed that the processor is rated for, and the “Core Speed” is the actual speed (or power) at which the CPU is running. In this particular case, the rated speed was 2.27 GHz and the actual speed was 1600.0 MHz (or ~1.56 GHz). This means that these CPUs were running at about 67% power. When we changed the power plan from balanced to high performance on this server, the CPU pressure immediately resolved and did not return.

Windows Server 2008 R2 introduced a new performance counter for tracking throttling of the CPU power: Processor Information\% of Maximum Frequency. This counter shows you the power level of the CPUs as the current percentage of the maximum frequency. If I had been tracking this performance counter for the above referenced SQL Server, this counter would have shown a value of approximate 67%.

You can change this setting via the Power settings in Control Panel on most servers. A few server models must have power configured in the BIOS, so if the CPUs are throttled while on the high performance plan, you may need to disable power throttling in the BIOS. Some server models can also override OS power setting via the BIOS. This makes it easy for infrastructure engineers to ensure that CPUs do not get throttled without having to interact with the operating system.

In addition to setting the power plan via the Control Panel, there are a couple of methods I have outlined via blog posts for setting the power plan programmatically. You can set it via the command line using the method outlined here: Enabling High Performance Power Plan Via Command Line (http://blog.idera.com/sql-server/performance-and-monitoring/enablinghighperformancepowerplanviacommandline/). Or you can set it via PowerShell as described here: Enabling High Performance Power Plan via PowerShell (http://www.sqlsoldier.com/wp/sqlserver/enabling-high-performance-power-plan-via-powershell).

There is no legitimate reason why CPUs for production SQL Servers should be power-throttled. All production servers should run in high performance mode all the time. If you are running with a balanced power plan to save energy, you are essentially throwing bags of money away because you can get better performance with fewer fully-powered CPUs than with a higher number of throttled CPUs. Changing this setting when your CPUs are throttled is basically free performance.

2 DEFAULT PARALLELISM SETTINGS

Parallelism is a very important feature of the SQL Server database engine. In general, it is a crucial element of getting good performance for larger queries. However, in certain scenarios, the parallelism settings can cause performance problems. The best way to handle these scenarios is to change the settings to more appropriate values. There are two instance-level configuration settings that you should consider changing from the default value: Maximum Degree of Parallelism (MaxDOP) and Cost Threshold for Parallelism.

It is a rare edge case where performance actually benefits from disabling parallelism. You will hear advice from time to time suggesting you should disable parallelism—often in response to high CXPacket waits. Be wary of any advice to disable parallelism (setting MaxDOP = 1).

CXPacket waits are a symptom of parallelism and are not necessarily a problem themselves (see "The Barking Dog Analogy" at http://www.sqlsoldier.com/wp/sqlserver/thebarkingdoganalogy). Your goal with parallelism is not to get rid of it—your goal is to find the right balance between parallelism and concurrency. It is not possible to know the exact best settings for parallelism without testing a real production workload. The best any of us can do is to provide guidelines as a starting point and then monitor and adjust as needed.

Most SQL Servers today are NUMA (non-uniform memory access) machines with a high number of CPUs designed to handle very large workloads. Conventional wisdom regarding the MaxDOP setting relies upon several conditional points that you must consider. The following guidelines are for total logical CPUs.

- If the machine is SMP (symmetric multiprocessing) and the CPU count is ≤ 8 CPUs, leave it at the default setting of 0.
- If the machine is SMP and the CPU count is > 8 CPUs, cap the setting at a maximum of 8.
- If the machine is NUMA, then cap the setting at the number of CPUs per NUMA node, or 8—whichever is less.

For most machines in use today, you mostly just need to focus on that last recommendation in the above guidelines. Over-parallelism can be a problem because it decreases concurrency and can cause queries to back up on worker threads waiting for time to execute on a CPU. For NUMA machines, over-parallelism is common if you do not restrict parallelism so that a query does not exceed the total number of CPUs on a single NUMA node. With few exceptions, SQL Server will opt to keep a parallelized task local within a single NUMA node when the number of parallel threads is high. This can lead to some threads of the same query sharing CPU time with other threads of the same query. If some CPU threads are handling multiple parallel threads of the same query, and some are handling only single threads, then the workload will be imbalanced—some threads may complete more quickly than others. Since all threads must complete for the entire parallel task to complete, some threads will be left sitting idle, thus causing CXPacket waits.

The other setting that you should consider changing is the "Cost Threshold for Parallelism." The default value for this setting is 5. Cost is an estimate of how much work will be involved to complete a particular task in a query plan. The calculation is based on the CPU ticks of the original developer's workstation at the time the feature was written. That was a long time ago, and the cost value alone is not very meaningful except as a way to gauge relative work.

However, we can use the cost estimates in query plans to understand whether or not a task is a candidate for parallelism. We can use these cost estimates to determine a threshold for small queries that we don't want to parallelize. With the default setting of 5, queries do not have to be very large to be a candidate for parallelism. Increasing this value will prevent parallelization of small queries and free up more CPU threads for processing other queries. The smaller queries will take a little more time to complete, but it will allow you to increase concurrency without reducing the MaxDOP for large queries.

On new SQL Servers that do not have a known workload which I can use to identify an appropriate value for this setting, I generally set cost threshold to 50. Experience has shown me that this is a good starting point for this setting and will only occasionally need further adjusting.

One of the best things you can do in terms of managing parallelism is to tune queries to reduce the amount of IO they must perform. This, of course, requires a lot more work than just setting the cost threshold and MaxDOP settings to appropriate values. For a quick performance gain, change these two settings.

3 DATA TYPE MISMATCHES

Causing implicit conversions in SQL Server can be very common if you're not careful.
Often, these conversions due to mismatched data types have a neglible cost. The amount of work required to simply convert a value to a different data type is generally very miniscule, and you cannot detect the difference in query cost. Other times, throwing a conversion into the mix can cause a table or index scan instead of a smaller, more efficient seek.

When there is a data type mismatch in the criteria of a query, in either the WHERE clause or a JOIN statement, the conversion may be applied to the data column rather than to a constant value. In order to ensure criteria are matched correctly, the entire column of data may have to be converted before values can be converted. I say "may have" because each version of SQL Server gets a little bit smarter about stuff like this.

To demonstrate this, I set up a demo using the AdventureWorksDW2012 sample database. I select a table with an nvarchar column and compared the differences between joining the column to a varchar column and to an nvarchar column. The table and column I chose were dbo.FactInternetSales.CarrierTrackingNumber.

First, I needed to create a couple of tables with data to use for the test, one with a varchar column and one with an nvarchar column. I also added an index to CarrierTrackingNumber so that an index seek would be an option the optimizer could choose.

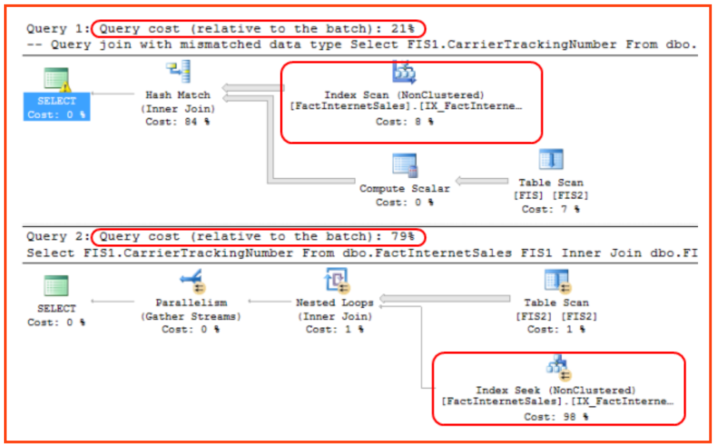
```sql
-- Set up tables for join
Select Cast(CarrierTrackingNumber as varchar(25)) As CTN
Into dbo.FIS
From dbo.FactInternetSales;

Select CarrierTrackingNumber As CTN -- nvarchar(25)
Into dbo.FIS2
From dbo.FactInternetSales;

-- Add an index that can be used for the query
Create Index IX_FactInternetSales_CarrierTrackingNumber
       On dbo.FactInternetSales(CarrierTrackingNumber);
```

The next step was to query the tables joined on the mismatched columns and the matched columns and compare the plans and execution times.

```sql
-- Query join with mismatched data type
Select FIS1.CarrierTrackingNumber
From dbo.FactInternetSales FIS1
Inner Join dbo.FIS FIS2
       On FIS1.CarrierTrackingNumber = FIS2.CTN;
```

```sql
-- Query join with matched data type
Select FIS1.CarrierTrackingNumber
From dbo.FactInternetSales FIS1
Inner Join dbo.FIS2 FIS2
       On FIS1.CarrierTrackingNumber = FIS2.CTN;
```

The query plans clearly show that the query with the mismatched data type has to perform an index scan of the index I added, whereas the query with matched data types can instead perform a seek. An oddity: when comparing the query plans to one another, it shows the optimizer thinks the first plan (with the mismatched data types and index scan) is more efficient than the second (with matched data types and an index seek). However, running the queries individually shows the truth. The query with mismatched data types and an index scan takes 2 minutes and 33 seconds. The query with the matched data types and an index seek takes less than a second.



```
Query 1: Query cost (relative to the batch): 21%
-- Query join with mismatched data type Select FIS1.CarrierTrackingNumber From dbo.
```

```
Query 2: Query cost (relative to the batch): 79%
Select FIS1.CarrierTrackingNumber From dbo.FactInternetSales FIS1 Inner Join dbo.FI
```

If we query the same table with a varchar string or variable instead of a join and examine the query plan, we see that SQL is actually smart enough to convert the constant value to nvarchar and perform the comparison rather than following the rules for conversion. Older versions of SQL Server may not make this same choice. As I said earlier, SQL Server gets smarter about things like these with every major release.

When you need to optimize a procedure or query, keep an eye out for data type mismatches. The query plan can be misleading about the impact that it has, but correcting these mismatches can deliver a huge boost to performance. I'm not advocating that you go out and immediately start scanning queries for data type mismatches, but when you do need to tune a query, it's an easy performance win. Or, if you are reviewing code for others, it's good to keep an eye out for these mismatches.

4 FUNCTIONS ON COLUMNS

Developers love code reusability. In fact, who doesn't love it? Over the years, however, DBAs and SQL Developers alike have also learned to hate functions. It does make developing easier when you need to include some complex piece of logic repeatedly in queries, but it often comes at a high cost.

There are several things wrong with applying functions to a column in queries. Whether it is in the criteria or in the SELECT clause, it can have a huge performance impact. When used in the criteria of a query (WHERE or JOIN clause), a function on a column can force an index scan instead of an index seek and execute the function one time for each row in the table. This behavior is similar to row-by-row looping execution. In other words—it's painful.

Functions can also wreak havoc with query plans because the cost of a function won't show up in query plans—and the plans may look like the function is presenting no problem at all. You can get a better idea of the true cost of the function by getting the plan for the code inside the function and then executing it in a loop equal to the number of records it will be expected to run against.

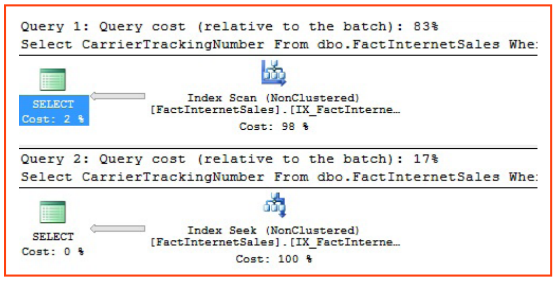
Some may think this applies only to user-defined functions, but even built-in SQL functions can have this problem. I see them used frequently without any concern to how their usage affects performance. Let's take a look at a simple example.

Once again, we will use the dbo.FactInternetSales table from the AdventureWorksDW2012 database. I will again use the CarrierTrackingNumber column that I created an index on when talking about data type mismatches. If you need to, recreate the same index for this demo form above.

Then, I compare plans for two versions of a common query pattern. One version uses the ISNULL function on both sides of the criteria so that, if the variable is null, the query returns records with null values and the other does multiple comparisons without using ISNULL (no function call on the column).

```
Declare @CTN nvarchar(25);

Set @CTN = NULL;

-- Function on column
Select CarrierTrackingNumber
From dbo.FactInternetSales
Where ISNULL(CarrierTrackingNumber, '') = ISNULL(@CTN, '');

-- Function on constant
Select CarrierTrackingNumber
From dbo.FactInternetSales
Where CarrierTrackingNumber = @CTN
Or (CarrierTrackingNumber Is Null And @CTN Is Null);
```

Both queries are functionally the same, but the plans are different. The query with the ISNULL functions performs a scan of the index and has a relative cost of 83% of the total cost of both queries. The query with the ISNULL functions performs a seek of the index and has a relative cost of 17%. The query with the function calls is almost 5 times more expensive.


```
Query 1: Query cost (relative to the batch): 83%
Select CarrierTrackingNumber From dbo.FactInternetSales Whe:
```
SELECT
Cost: 2 %
Index Scan (NonClustered)
[FactInternetSales].[IX_FactInterne…
Cost: 98 %
```
Query 2: Query cost (relative to the batch): 17%
Select CarrierTrackingNumber From dbo.FactInternetSales Whe:
```
SELECT
Cost: 0 %
Index Seek (NonClustered)
[FactInternetSales].[IX_FactInterne…
Cost: 100 %

That difference of almost 5 times the cost also plays out in the IO statistics. If I re-execute the same queries, this time with STATISTICS IO set on, the IO for the query with the function calls is almost 5 times as much as the query without the function calls. Logical reads is 430—versus 92 pages. Imagine how this issue would be compounded exponentially as the data set grows.

```
(25398 row(s) affected)
Table 'FactInternetSales'. Scan count 1, logical reads 430, physical reads 0, read-
ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

(25398 row(s) affected)
Table 'FactInternetSales'. Scan count 1, logical reads 92, physical reads 0, read-
ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
```

Like the issue I talked about earlier, SQL Server has become—and continues to get—much smarter about these scenarios. For simple queries, you may not see a difference, but the more complex a query is, the greater the chance that even a built-in function will adversely affect the performance.

The task here is simple. When you are tuning a query or procedure, this is another design pattern to look for in the query plans and queries. When you find instances of functions—user-defined or built-in system—rewrite the query to so that the function call is not on a column. Removing the column altogether also works very well.

5 MAX SERVER MEMORY AT DEFAULT SETTING

The Max Server Memory setting is responsible for a wide variety of problems. Many people are surprised at how many things can perform poorly when free memory is low and there isn't enough memory for things to operate as they normally do. This is one of the first things I think of when someone complains how a process that uses memory not in the buffer pool normally runs fast, but occasionally runs really long. In particular, what I hear is that when it is running slow, restarting SQL Server "fixes" the problem for a while, but it returns after a while.

Restarting SQL Server flushes and releases all memory back to the operating system. When SQL starts up, it begins using memory again and, if the amount of data and workload is sufficient, it will keep using more memory until it is all gone or it has reached its maximum allowable memory. SQL Server was designed to assume it is the most important application on the server, and it does not give up memory willingly once it has it. When the operating system tells SQL Server it needs more, SQL is not always very responsive with how quickly it frees memory, and it only frees up a little at a time until the OS stops bugging it.

In SQL Server 2012, memory allocations were changed so that the buffer pool, the amount of memory covered by the max server memory setting, allocates more objects. The need for external free memory is lessened somewhat in SQL 2012, but there is still plenty that needs external memory. SSIS packages and linked server queries are two things that require external memory, and you can see processes that use these suffer very poor performance when free memory is low.

One of the first things I check when I inherit an existing server is the max server memory setting, especially when that server is known to have unexplained performance problems. For new servers, I make sure the memory settings are configured before releasing it for usage.

You may be wondering at this point, "What amount is the right setting for max memory?" That question depends heavily on many factors. In general though, the best way to figure this, is to determine how much memory you need to keep free for the OS and other external processes, subtract that number from the total memory, and use the resulting number as your max server memory setting.

One way to figure out the best option for this setting is to set it to a value much lower than you think is needed. Then monitor free memory on the server (Memory\Available MBytes) to figure out exactly how much memory the additional processes are going to use. Once you know this, you know how high you can bump the max server memory.

There are some good rule-of-thumb recommendations for unknown workloads as well. You can start with the numbers listed below and then pad on additional memory amounts for higher CPUs counts or for external processes and components that will be used on the server.

- 4 GB server with up to 2 CPUs - a minimum of 1 GB of RAM
- 8 GB or more with up to 4 CPUs - a minimum of 2 GB of RAM
- 32 GB or more with up to 16 CPUs - a minimum of 4 to 6 GB of RAM
- 64 GB or more with up to 32 CPUs – a minimum of 8 to 10 GB of RAM

Even if all you choose to do is apply the rule-of-thumb method above, it's a much better alternative than leaving the max server memory setting at the default value, which ultimately equates to all memory. You can dive deep into setting this option, but you can also make a quick win and simply set it to an estimate of what you think is the right value. Even an estimate being a little off will be less troublesome than the default setting.

CONCLUSION

The tips I discuss in this paper will help you both deal with and head off some very common problems database administrators and SQL developers struggle with every day. You should be proactive; set the power plan, parallelism settings, and max server memory to more appropriate settings immediately. **Don't wait for the problems to occur. Fix them today.**

The issues with the data type mismatches and functions on columns are things you should watch for any time you deal with SQL code. If you are writing SQL code, tuning it, or troubleshooting poor performance, you should look for these known problems to get big performance wins with minimal effort.

TRY IDERA'S SQL DIAGNOSTIC MANAGER

24X7 SQL PERFORMANCE MONITORING, ALERTING AND DIAGNOSTICS

- **Performance monitoring** for physical and virtual SQL Servers
- **Deep query analysis** to identify excessive waits and resource consumption
- **History browsing** to find and troubleshoot past issues
- **Adaptive & automated alerting** with 100+ pre-defined and configurable alerts
- **Capacity planning** to see database growth trends and minimize server sprawl
- **SCOM management pack** for integration with System Center

TRY IT FREE FOR 14 DAYS.