Ten Tips for Optimizing SQL Server Performance

Written by Patrick O'Keefe and Richard Douglas

Abstract

Performance optimization on SQL Server can be challenging. A vast array of information is available on how to address performance problems in general. However, there is not much detailed information on specific issues and even less information on how to apply that specific knowledge to your own environment.

This paper offers 10 tips for optimizing SQL Server performance, with SQL Server 2008 and 2012 in mind. There are no definitive lists of the most important tips, but you won't go wrong by starting here.

As a DBA, you undoubtedly have your own perspective and will your own favorite optimization tips, tricks and scripts. Why not join the discussion on SQLServerPedia?

Introduction

Consider your tuning goals.

We all want to get the most value from our SQL Server deployments. Increasing the efficiency of your database servers frees up system resources for other tasks, such as business reports or ad hoc queries. To get the most from your organization's hardware investment, you need to ensure the SQL or application workload running on the database servers is executing as quickly and efficiently as possible.

But how you tune depends on your goals. You might be asking, "Am I getting the best efficiency from my SQL Server deployment?" But someone else might be asking, "Will my application scale?" Here are the main ways of tuning a system:

- Tuning to meet service level agreement (SLA) or key performance indicator (KPI) targets
- Tuning to improve efficiency, thereby freeing up resources for other purposes
- Tuning to ensure scalability, thereby helping to maintain SLAs or KPIs in the future

Work to maximize the scalability and efficiency of all database servers—even if business requirements are currently being met.

Remember that tuning is an ongoing process, not a one-time fix.
Performance optimization is a continual process. For instance, when you tune for SLA targets, you can be "finished." However, if you are tuning to improve efficiency or to ensure scalability, your work is never truly complete; this type of tuning should be continued until the performance is "good enough." In the future, when the performance of the application is no longer good enough, the tuning cycle should be performed again.

"Good enough" is usually defined by business imperatives, such as SLAs or system throughput requirements. Beyond these requirements, you should be motivated to maximize the scalability and efficiency of all database servers—even if business requirements are currently being met.

About this document
Performance optimization on SQL Server is challenging. There is a wealth of generalized information on various data points—for example, performance counters and dynamic management objects (DMOs)—but there is very little information on what to do with this data and how to interpret it. This paper describes 10 important tips that will be useful in the trenches, allowing you to turn some of this data into actionable information.

Tip #10. The Baselining and Benchmarking methodology helps you spot problems.

Overview of the baselining and benchmarking process
Figure 1 illustrates the steps in the baselining process. The following sections will explain the key steps in the process.

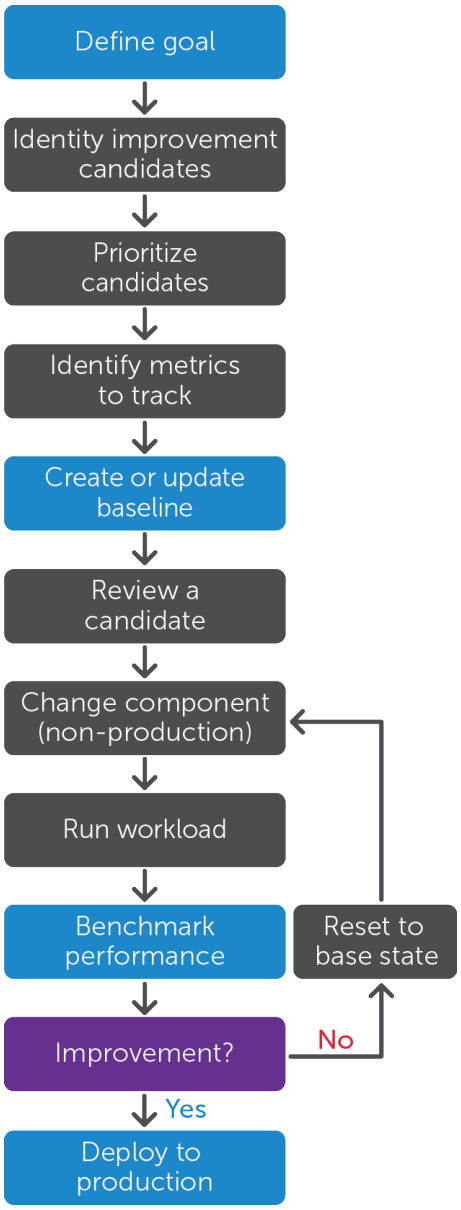

Figure 1. The baselining process

Before you start making changes, get a record of current system performance.
When tuning and optimizing, most of us will be tempted to jump straight in and start tweaking. Have you ever collected your car from a mechanic and wondered if it's actually performing worse than it did before? You would like to complain—but you just don't know for sure. You might also wonder whether the cause of the problem you think you see now is something the mechanic did, or something that happened afterward. Unfortunately, database performance can suffer from the same problems.

After reading this paper, you will be full of ideas and want to implement your baseline and benchmarking strategy immediately. The first step you need to take is not the most exciting, but it is certainly one of the most important: it is time to establish just how well your environment measures up against the criteria you plan to change.

Determine your goals.
Before you do anything to your system, decide what you want to achieve. Are there aspects of your SLA that need to be addressed with respect to performance, resource consumption or capacity? Are you addressing a current issue in production? Have there been complaints with respect to resource time? Set some clear goals.

Most of us have many databases and instances to look after. In order to maximise the value from our efforts, we need to think carefully what is required from a particular system in order for it to perform well and meet user expectations. If you over-do the analysis and tuning, you can find that a disproportionate amount of time is spent on low-priority systems to the detriment of your core production systems. Be clear on precisely what you want to achieve from your measurement and tuning efforts. Prioritize these and ideally get buy-in and agreement from a business sponsor.

Establish the norm.
Once you've targeted what you want to achieve, then you need to decide how you will measure success. What are the OS counters, SQL Server counters, resource measures and other data points that will give you the required insight?

Once you have that list, you need to establish your baseline, or the typical performance of your system against your chosen criteria. You need to collect enough data over a long enough period of time to give a representative sample of the typical performance of the system. Once you have the data, you can average the values over the period to establish your first baseline. After you make modifications to your system, you'll take a new benchmark and compare it to the original one so you can objectively measure the effect of your changes.

Don't track just the average values; also track the deviation from the norm.
It 's important to be careful of averages, though. At the very least, calculate the standard deviation of each counter to give you an indication of the variation over time. Consider the rock climber who is told that the average diameter of the rope is 1 cm. The climber confidently jumps over the side. He dangles several hundred feet over the sharp rocks and smiles smugly. Then he's told that the thickest section of the rope is 2 cm and the thinnest is 0.1cm. Oops!

If you are unfamiliar with standard deviation, grab a beginner's book on statistics. You don't need to get too sophisticated, but it helps to know the basics.

The key message here is not just to track averages; also track the deviation from the norm (mean). Decide what that norm should be (this is often found in your SLA). Your mission is not to get the maximum possible performance, but to achieve the best possible fit for your performance goals and then to limit the deviation from these goals

Benchmarking helps you spot abnormal behavior because you have a good indication of what normal behavior is.

Your mission is not to get the maximum possible performance; it is to achieve the best possible fit for your performance goals and then to limit the deviation from these goals as much as possible.

as much as possible. Anything more represents wasted time and effort and could also point to under-utilization of infrastructural resources.

How much data is needed for a baseline?
The amount of data required to establish a baseline depends on the extent to which your load varies over time. Talk to system administrators, end users and application administrators. They will generally have a good feel for what the usage patterns are. You should gather enough data to cover off-peak, average and peak periods.

It is important to gauge the amount which load varies and how frequently. Systems with predictable patterns will require less data. The more variation, the smaller the measurement interval and the longer you will need to measure to develop a reliable baseline. To take our climbing analogy a little further, the longer the length of rope you examine, the better the chance that you will spot the variations. How critical the system is and the impact of its failure will also affect how much scrutiny it must undergo and how reliable the sample must be.

Storing the data
The more parameters you track and the smaller the frequency, the larger the data set you will collect. It may seem obvious, but you do need to consider the capacity that will be required to store your measurement data. Once you have some data, it should be pretty straightforward to extrapolate the extent to which this repository will grow over time. If you are going to monitor over an extended period, think about aggregating historical data at intervals to prevent the repository from growing too large.

For performance reasons, your measurement repository should not reside in the same place as the database you are monitoring.

Limit the number of changes you make at one time.
Try to limit the number of changes you make between each benchmark. Construct your modifications to test one particular hypothesis. This will allow you to meticulously rule in or out each improvement candidate. When you do hone in on a solution, you will understand exactly why you are seeing the change in behaviour, and this will often reveal a series of additional potential improvement options.

Analyzing the data
Once you have made changes to your system, you will want to determine whether they have had the desired effect. To achieve this, repeat the measurements you took for your original baseline, over a similarly representative time scale. You can then compare the two baselines to:

- **Determine whether your changes had the desired effect**—When you tweak a configuration setting, optimize an index, or some change SQL code, the baseline enables you to tell whether that change had the desired effect. If you receive a complaint about slower performance, you can say for sure whether the statement is accurate from the perspective of the database.

The most frequent mistake that most junior DBAs will make is to jump to conclusions too early. Often you will see someone jump for joy as they observe an instant and observable increase in performance after they make one or more changes. They deploy to production and rush to send emails declaring the issue resolved. But the celebrations can be short-lived when the same issues re-emerge sometime after or some unknown side effect causes another issue. Quite often this can result in a state that is less desirable than the original. Once you think you have found the answer to an issue, test it and benchmark the results against your baseline. This is the only reliable way of being sure you have made progress.

- **Determine whether a change introduced any unexpected side effects**—A baseline also enables you to see objectively whether a change affected a counter or measure that you hadn't expect it to affect.
- **Anticipate problems before they happen**—Using a baseline, you can establish accurate performance norms against typical load conditions. This enables you to predict whether and when you will hit problems in the future, based on how resource consumption is trending today or based on projected workloads for future scenarios. For example, you perform capacity planning: by extrapolating from current typical resource consumption per connected user, you can predict when your systems will hit user connection bottlenecks.
- **Troubleshoot problems more effectively**—Ever spent several long days and nights fire-fighting a performance issue with your database only to find it was actually nothing to do with the database itself? Establishing a baseline makes it far more straightforward to eliminate your database instance and target the culprit. For example, suppose memory consumption has suddenly spiked. You notice the number of connections increasing sharply and it is well above your baseline. A quick call to the application administrator confirms that a new module has been deployed in the eStore. It doesn't take long to establish that that the new junior developer is writing code that doesn't release database connections as it should. I bet you can think of many more scenarios just like this.

  Ruling out the things that are NOT responsible for a problem can save a great deal of time by clearing the clutter and providing a laser focus on exactly what is causing the issue. There are lots of examples where we can compare system counters to SQL server counters to quickly rule the database in or out of an issue. Once the usual suspects are ruled out you can now begin search for significant deviations from baseline, collect related indicators together and begin to drill into the root cause.

Repeat the baselining process as often as needed.
Good tuning is an iterative and scientific process. The tips presented in this document provide a really good starting point, but they are just that: a starting point. Performance tuning is highly personalised and is governed by the design, makeup and usage of each individual system.

The baselining and benchmarking methodology is the cornerstone of good and reliable performance tuning. It provides the map, a reference and the waypoints we require to figure out where we need to go and how to get there, to help make sure we don't get lost on the way. A structured approach allows us to build reliable and consistent performance across our database portfolio.

The amount of data required to establish a baseline depends on the extent to which your load varies over time.

Tip #9. Performance counters give you quick and useful information about currently running operations.

Reasons for monitoring performance counters

A very common question related to SQL Server performance optimization is: "What counters should I monitor?" In terms of managing SQL Server, there are two broad reasons for monitoring performance counters:

- Increasing operational efficiency
- Preventing bottlenecks

Although they have some overlap, these two reasons allow you to easily choose a number of data points to monitor.

Monitoring performance counters to improve operational efficiency

Operational monitoring checks for general resource usage. It helps answer questions like:

- Is the server about to run out of resources like CPU, disk space, or memory?
- Are the data files able to grow?
- Do fixed-size data files have enough free space for data?

Limit the number of changes you make between each benchmark, so you can meticulously evaluate the effects of each change.

You can also use the data for trending purposes. A good example would be collecting the sizes of all the data files in order to trend their growth rates and forecast future resource requirements.

To answer the three questions posed above, you should look at the following counters:

Counter	Enables you to
Processor\%Processor Time	Monitor the CPU consumption on the server
LogicalDisk\Free MB	Monitor the free space on the disk(s)
MSSQL$Instance:Databases\DataFile(s) Size (KB)	Trend growth over time
Memory\Pages/sec	Check for paging, which is a good indication that memory resources might be short
Memory\Available MBytes	See the amount of physical memory available for system use

Monitoring performance counters to prevent bottlenecks

Bottleneck monitoring focuses more on performance-related matters. The data you collect helps answer questions such as:

- Is there a CPU bottleneck?
- Is there an I/O bottleneck?
- Are the major SQL Server subsystems, such as the buffer cache and plan cache, healthy?
- Do we have contention in the database?

To answer these questions, look at the following counters:

Counter	Enables you to
Processor\%Processor Time	Monitoring CPU consumption allows you to check for a bottleneck on the server (indicated by high sustained usage).
High percentage of Signal Wait	Signal wait is the time a worker spends waiting for CPU time after it has finished waiting on something else (such as a lock, a latch or some other wait). Time spent waiting on the CPU is indicative of a CPU bottleneck. Signal wait can be found by executing DBCC SQLPERF(waitstats) on SQL Server 2000 or by querying sys.dm_os_wait_stats on SQL Server 2005.
Physical Disk\Avg. Disk Queue Length	Check for disk bottlenecks: if the value exceeds 2 then it is likely that a disk bottleneck exists.
MSSQL$Instance:Buffer Manager\Page Life Expectancy	Page Life Expectancy is the number of seconds a page stays in the buffer cache. A low number indicates that pages are being evicted without spending much time in the cache, which reduces the effectiveness of the cache.
MSSQL$Instance:Plan Cache\Cache Hit Ratio	A low Plan Cache hit ratio means that plans are not being reused.
MSSQL$Instance:General Statistics\Processes Blocked	Long blocks indicate contention for resources.

Tip #8. Changing server settings can provide a more stable environment.

Changing settings within a product to make it more stable may sound counter intuitive, but in this case it really does work. As a DBA, your job is to ensure a consistent level of performance for your users when they request data from their applications. Without changing the settings outlined in the rest of this document, you may experience scenarios that can degrade performance for your users without warning. These options can easily be found in sys.configurations, which lists server level configurations, available along with extra information. The Is_Dynamic attribute in sys.configurations shows whether the SQL Server instance will need to be restarted after making a configuration change. To make the change you would call the sp_configure stored procedure with the relevant parameters.

The Min and Max memory settings can guarantee a certain level of performance.

Suppose we have an Active/Active cluster (or indeed a single host with multiple instances). We can make certain configuration changes that can guarantee we can meet our SLAs in the event of a failover where both instances would reside on the same physical box.

In this scenario, we would make changes to both the Min and Max memory setting to ensure that the physical host has enough memory to cope with each instance without having to constantly try to aggressively trim the working set of the other. A similar configuration change can be made to use certain processors in order to guarantee a certain level of performance.

It is important to note that setting the maximum memory is not just suitable for instances on a cluster, but also those instances that share resources with any other application. If SQL Server memory usage is too high, the operating system may aggressively trim the amount of memory it can use in order to allow either itself or other applications room to operate.

- **SQL Server 2008**—In SQL Server 2008 R2 and before, the Min and Max memory setting would restrict only the amount of memory that the buffer pool uses – more specifically only single 8KB page allocations. This meant if you ran processes outside of the buffer pool (such as extended stored procedures, CLR or other components such as Integration Services, Reporting Services or Analysis Services), you would need to reduce this value further.
- **SQL Server 2012**—SQL Server 2012 changes things slightly as there is a central memory manager. This memory manager now incorporates multi-page allocations such as large data pages and cached plans that are larger than 8KB. This memory space now also includes some CLR functionality.

Two server options can indirectly help performance.

There are no options that directly aid performance but there are two options that can help indirectly.

- **Backup compression default**—This option sets backups to be compressed by default. Whilst this may churn up extra CPU cycles during compression, in general less CPU cycles are used overall when compared to an uncompressed backup, since less data is written to disk. Depending on your I/O architecture, setting this option may also reduce I/O contention.
- The second option may or may not be divulged in a future tip on the plan cache. You'll have to wait and see if it made our top 10 list.

Sage advice about troubleshooting in just three words: "Eliminate or incriminate."

Monitoring performance counters can help you increase operational efficiency and prevent bottlenecks.

Tip #7. Find rogue queries in the plan cache.

Once you have identified a bottleneck, you need to find the workload that is causing the bottleneck. This is a lot easier to do since the introduction of Dynamic Management Objects (DMOs) in SQL Server 2005. Users of SQL Server 2000 and prior will have to be content with using Profiler or Trace (more on that in Tip #6).

Diagnosing a CPU bottleneck

In SQL Server, if you identified a CPU bottleneck, the first thing that you would want to do is get the top CPU consumers on the server. This is a very simple query on sys.dm_exec_query_stats:

```
SELECT TOP 50
      qs.total_worker_time / execution_count AS avg_worker_time,
      substring (st.text, (qs.statement_start_offset / 2) + 1,
          ( (  CASE qs.statement_end_offset WHEN -1
                 THEN datalength (st.text)
               ELSE qs.statement_end_offset END
             - qs.statement_start_offset)/ 2)+ 1)
        AS statement_text,
      *
  FROM sys.dm_exec_query_stats AS qs
      CROSS APPLY sys.dm_exec_sql_text (qs.sql_handle) AS st
ORDER BY
avg_worker_time DESC
```

The really useful part of this query is your ability to use cross apply and sys.dm_exec_sql_text to get the SQL statement, so you can analyze it.

Diagnosing an I/O bottleneck

It is a similar story for an I/O bottleneck:

```
SELECT TOP 50
      (total_logical_reads + total_logical_writes) AS total_logical_io,
      (total_logical_reads / execution_count) AS avg_logical_reads,
      (total_logical_writes / execution_count) AS avg_logical_writes,
      (total_physical_reads / execution_count) AS avg_phys_reads,
      substring (st.text,
              (qs.statement_start_offset / 2) + 1,
              ((CASE qs.statement_end_offset WHEN -1
                  THEN datalength (st.text)
                ELSE qs.statement_end_offset END
                - qs.statement_start_offset)/ 2)+ 1)
              AS statement_text,
       *
  FROM sys.dm_exec_query_stats AS qs
      CROSS APPLY sys.dm_exec_sql_text (qs.sql_handle) AS st
ORDER BY total_logical_io DESC
```

Tip #6. SQL Profiler is your friend.

Understanding SQL Server Profiler and Extended Events

SQL Server Profiler is a native tool that ships with SQL Server. It allows you to create a trace file that captures events occurring in SQL Server. These traces can prove invaluable in providing information about your workload and poorly performing queries. This whitepaper is not going to dive into details on how to use the Profiler tool. For information on how to use SQL Server Profiler, check out the video tutorial on SQLServerPedia.

Whilst it is true that SQL Server Profiler has been marked as deprecated in SQL Server 2012 in favor of Extended Events, it should be noted that this is just for the database engine and not for SQL Server Analysis Services. Profiler can still provide great insight into the working of applications in real time for many SQL Server environments.

Using Extended Events is beyond the scope of this whitepaper; for a detailed description of Extended Events see the whitepaper, "How to Use SQL Servers Extended Events and Notifications to Proactively Resolve Performance Issues". Suffice to say that Extended Events were introduced in SQL Server 2008 and updated in SQL Server 2012 to include more events and an eagerly anticipated UI.

Please note that running Profiler requires the ALTER TRACE permission.

How to use SQL Profiler

Here's how to build on the process of collecting data in Performance Monitor (Perfmon) and correlate the information on resource usage with data on the events being fired inside SQL Server:

1. Open Perfmon.
2. If you do not have a Data Collector set already configured create one now using the advanced option and the counters from Tip 9 as a guide. Do not start the data collector set yet.
3. Open Profiler.
4. Create a new trace by specifying details about the instance, events and columns you want to monitor, as well as the destination.
5. Start the trace.
6. Switch back to Perfmon to start the data collector set.
7. Leave both sessions running until the required data has been captured.
8. Stop the Profiler trace. Save the trace and then close it.
9. Switch to Perfmon and stop the data collection set.
10. Open the recently saved trace file in Profiler.
11. Click File and then Import Performance Data.
12. Navigate to the Data Collection data file and select the performance counters of interest.

You will now be able to see the performance counters in conjunction with the Profiler trace file (see Figure 2), which will enable a much faster resolution of bottlenecks.

Extra tip: The steps above use the client interface; to save resources, a server-side trace would be more efficient. Please refer to Books Online for information about how to start and stop server-side traces.

Operational monitoring checks for general resource usage. Bottleneck monitoring focuses more on performance-related matters.

The Is_Dynamic attribute in Sys.Configurations shows whether the SQL Server instance will need to be restarted after making a configuration change.

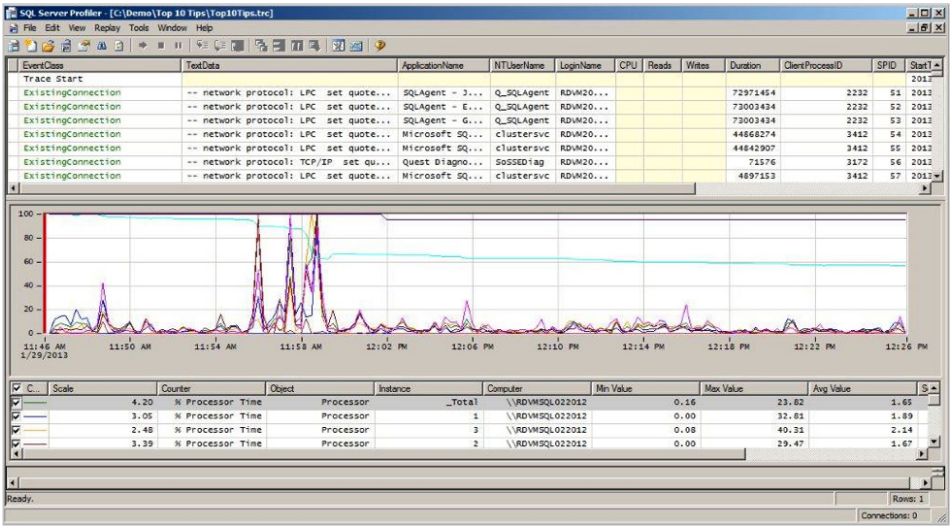

Figure 2. A correlated view of performance counters in conjunction with the Profiler trace file

Tip #5. Configure SANs for SQL Server performance.

Storage area networks (SANs) are fantastic. They offer the ability to provision and manage storage in a simple way. SANs can be configured for fast performance from a SQL Server perspective—but they often aren't. Organizations usually implement SANs for reasons such as storage consolidation and ease of management—not for performance. To make matters worse, generally you do not have direct control over how the provisioning is done on a SAN. Thus, you will often find that the SAN has been configured for one logical volume where you have to put all the data files.

Best practices for configuring SANs for I/O performance

Having all the files on a single volume is generally not a good idea if you want the best I/O performance. As best practices, you will want to:

- **Place log files on their own volume**, separate from data files. Log files are almost exclusively written to sequentially and not read (exceptions to this include Database Mirroring and Always On Availability Groups). You should always configure for fast write performance.
- **Place tempdb on its own volume**. Tempdb is used for myriad purposes by SQL Server internally, so having it on its own I/O subsystem will help. To further fine tune performance, you will first need some stats.
- **Consider creating multiple data files and filegroups in VLDBs** in order to benefit from parallel I/O operations.
- **Place backups on their own drives** for redundancy purposes as well as to reduce I/O contention with other volumes during maintenance periods.

Collecting data

There are, of course, the Windows disk counters, which will give you a picture of what Windows thinks is happening. (Don't forget to adjust raw numbers based on RAID configuration.) SAN vendors often provide their own performance data.

SQL Server also provides file-level I/O information:

- **Versions prior to SQL 2005**—Use the function fn_virtualfilestats.
- **Later versions**—Use the dynamic management function sys.dm_io_virtual_file_stats.

By using this function in the following code, you can:

- Derive I/O rates for both reads and writes
- Get I/O throughput
- Get average time per I/O
- Look at I/O wait times

```
SELECT db_name (a.database_id) AS [DatabaseName],
      b.name AS [FileName],
      a.File_ID AS [FileID],
      CASE WHEN a.file_id = 2 THEN 'Log' ELSE 'Data' END AS [FileType],
      a.Num_of_Reads AS [NumReads],
      a.num_of_bytes_read AS [NumBytesRead],
      a.io_stall_read_ms AS [IOStallReadsMS],
      a.num_of_writes AS [NumWrites],
      a.num_of_bytes_written AS [NumBytesWritten],
      a.io_stall_write_ms AS [IOStallWritesMS],
      a.io_stall [TotalIOStallMS],
      DATEADD (ms, -a.sample_ms, GETDATE ()) [LastReset],
      ( (a.size_on_disk_bytes / 1024) / 1024.0) AS [SizeOnDiskMB],
      UPPER (LEFT (b.physical_name, 2)) AS [DiskLocation]
  FROM sys.dm_io_virtual_file_stats (NULL, NULL) a
      JOIN sys.master_files b
        ON a.file_id = b.file_id AND a.database_id = b.database_id
ORDER BY a.io_stall DESC;
```

Setting backups to be compressed by default reduces I/O contention.

Analyzing the data

Please pay special attention to at the "LastReset" value in the query; it shows the last time the SQL Server service was started. Data from Dynamic Management Objects are not persisted, so any data that is being used for tuning purposes should be validated against how long the service has been running; otherwise, false assumptions can be made.

Using these numbers, you can quickly narrow down which files are responsible for consuming I/O bandwidth and ask questions such as:

- Is this I/O necessary? Am I missing an index?
- Is it one table or index in a file that is responsible? Can I put this index or table in another file on another volume?

Tips for tuning your hardware

If the database files are placed correctly and all object hotspots have been identified and separated onto different volumes, then it is time to take a close look at the hardware.

Tuning hardware is a specialist topic outside of the scope of this whitepaper. There are a few best practices and tips that I can share with you which will make this easier, though:

- Do not use the default allocation unit size when creating volumes for use with SQL Server. SQL Server uses 64KB extents, so this value should be the minimum.
- Check to see if your partitions are correctly aligned. Jimmy May has written a fantastic whitepaper on this subject. Misaligned partitions can reduce performance by up to 30 percent.
- Benchmark your system's I/O using a tool such as SQLIO. You can watch a tutorial on this tool.

DatabaseName	FileName	FileID	FileType	NumReads	NumBytesRead	IOStallReadsMS	NumWrites	NumBytesWritten	IOStallWritesMS	TotalIOStall...	LastReset	SizeOnDiskMB	DiskLocation
AWTarget2012	AdventureWorks2012_Data	1	Data	91	5709824	10626	8	73728	2	10628	31/01/201...	205.000000	C:
NewTest	NewTest	1	Data	36	2113536	9963	1	8192	19	9982	31/01/201...	5.000000	C:
AWSource2012	AdventureWorks2012_Data	1	Data	84	5251072	9731	2	16384	0	9731	31/01/201...	205.000000	C:
AW20121	AdventureWorks2012_Data	1	Data	74	4677632	9151	1	8192	0	9151	31/01/201...	205.000000	C:

Figure 3. File-level I/O information from SQL Server

SQL Profiler can still provide great insight into the working of applications in real time for many SQL Server environments.

Tip #4.Cursors and other bad T-SQL frequently return to haunt applications.

An example of bad code

At a previous job, I found what has to be the worst piece of code I have ever seen in my career. The system has long since been replaced, but here is an outline of the process the function went through:

1. Accept parameter value to strip.
2. Accept parameter expression to strip.
3. Find the length of the expression.
4. Load expression into a variable.
5. Loop through each character in the variable and check to see if this character matched one of the values to strip. If it does update the variable to remove it.
6. Continue to next character until expression has been fully checked.

If you have a look of horror on your face, then you are in good company. I am of course explaining somebody's attempt to write their own T-SQL "REPLACE" statement!

The worst part was that this function was used to update addresses as part of a mailing routine and would be called tens of thousands of times a day.

Running a server-side profiler trace will enable you to view your server's workload and pick out frequently executed pieces of code (which is the way this "gem" was found).

Query tuning using query plans

Bad T-SQL can also appear as inefficient queries that do not use indexes, mostly because the index is incorrect or missing. It's vitally important to have a good understanding of how to tune queries using query plans.

A detailed discussion of query tuning using query plans is beyond the scope of this white paper. However, the simplest way to start this process is by turning SCAN operations into SEEKs. SCANs read every row in the table or index, so, for large tables, SCANs are expensive in terms of I/O. A SEEK, on the other hand, will use an index to go straight to the required row, which, of course, requires an index. If you find SCANs in your workload, you could be missing indexes.

There are a number of good books on this topic, including:

- "Professional SQL Server Execution Plan Tuning" by Grant Fritchey
- "Professional SQL Server 2012 Internals & Troubleshooting" by Christian Bolton, Rob Farley, Glenn Berry, Justin Langford, Gavin Payne, Amit Banerjee, Michael Anderson, James Boother, and Steven Wort
- "T-SQL Fundamentals for Microsoft SQL Server 2012 and SQL Azure" by Itzik Ben-Gan

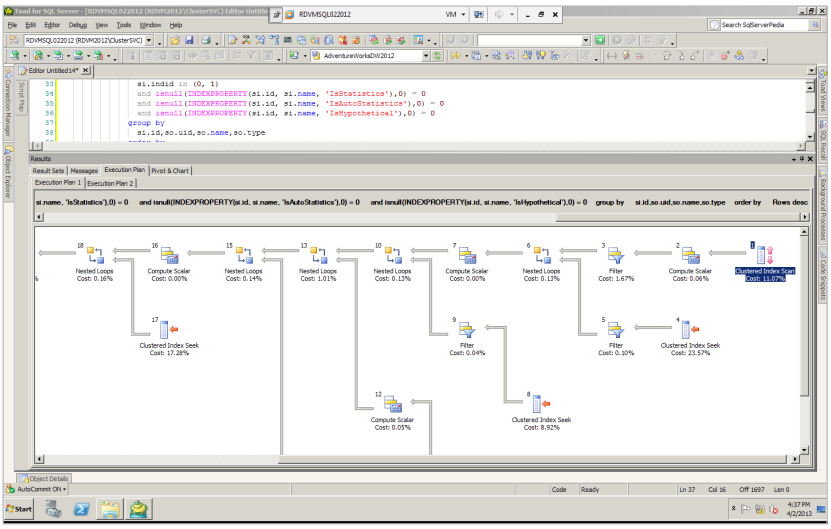

Figure 4. Example of a large query plan

Tip #3. Maximize plan reuse for better SQL Server caching.

Why reusing query plans is important

Before executing a SQL statement, SQL Server first creates a query plan. This defines the method SQL Server will use to take the logical instruction of the query and implement it as a physical action against the data.

Creating a query plan can require significant CPU. Thus, SQL Server will run more efficiently if it can reuse query plans instead of creating a new one each time a SQL statement is executed.

Evaluating whether you are getting good plan reuse

There are some performance counters available in the SQL Statistics performance object that will tell you whether you are getting good plan reuse. This formula tells you the ratio of batches submitted to compilations:

You want this number to be as small as possible. A 1:1 ratio means that every batch submitted is being compiled, and there is no plan reuse at all.

> SANs can be configured for fast performance from a SQL Server perspective—but they often aren't.

```
(Batch Requests/sec – SQL Compilations/sec) / Batch Requests/sec
```

Addressing poor plan reuse

It's not easy to pin down the exact workload that is responsible for poor plan reuse, because the problem usually lies in the client application code that is submitting queries. Therefore, you may need to look at the client application code that is submitting queries.

To find code that is embedded within a client application, you will need to use either Extended Events or Profiler. By adding the SQL:StmtRecompile event into a trace, you will be able to see when a recompile event occurs. (There is also an event called SP:Recompile; this event is included for backwards compatibility since the occurrence of recompilation was changed from the procedure level to the statement level in SQL Server 2005.)

A common problem is that the code is not using prepared parameterized statements. Using parameterized queries not only improves plan reuse and compilation overhead, but it also reduces the SQL injection attack risk involved with passing parameters via string concatenation. Figure 5 shows two code examples. Though they are contrived, they illustrate the difference between building a statement through string concatenation and using prepared statements with parameters.

File-level I/O information from SQL Server can help you pinpoint which files are consuming I/O bandwidth.

```
public void ExecuteSomeSQL(int aParam) {
    //cmd is a SqlCommand created somewhere else

    cmd.CommandType = CommandType.Text;
    cmd.CommandText = "select foo1, foo2, foo3 from bar where foo1=" + aParam.ToString();

    SqlDataReader dr = cmd.ExecuteReader();
    try {
        while (dr.Read()) {

        }
    } finally {
        dr.Close();
    }

}
```

Bad

```
public void ExecuteSomeSQL(int aParam) {
    //cmd is a SqlCommand created somewhere else

    cmd.CommandType = CommandType.Text;
    cmd.CommandText = "select foo1, foo2, foo3 from bar where foo1 = @foo1";

    cmd.Parameters["@foo1"].Value = aParam;

    SqlDataReader dr = cmd.ExecuteReader();
    try {
        while (dr.Read()) {

        }
    } finally {
        dr.Close();
    }

}
```

Good

Figure 5. Comparing code that builds a statement through string concatenation and code that uses prepared statements with parameters

SQL Server cannot reuse the plan from the "bad" example in Figure 5. If a parameter had been a string type, this function could be used to mount a SQL injection attack. The "good" example is not susceptible to a SQL injection attack because a parameter is used, and SQL Server is able to reuse the plan.

The missing configuration setting from Tip #8

Those of you with a good memory will remember that in Tip #8 (where we talked about configuration changes), there was one more piece of advice left to bestow.

SQL Server 2008 introduced a configuration setting called "Optimize for ad hoc workloads." Setting it will tell SQL Server to store a stub plan rather than a full plan in the plan cache. This is especially useful for environments that use dynamically built T-SQL code or Linq which may result in the code not being re-used.

The memory allocated to the plan cache resides in the buffer pool. Therefore, a bloated plan cache reduces the amount of data pages that can be stored in the buffer cache–so there will be more round trips to fetch data from the I/O subsystem, which can be very expensive.

Tip #2. Learn how to read the SQL Server buffer cache and minimize cache thrashing.

Why the buffer cache matters

As alluded to in the rather elegant segue above, the buffer cache is a large area of memory used by SQL Server to reduce the need to perform physical I/O. No SQL Server query execution reads data directly off the disk; the database pages are read from the buffer cache. If the sought-after page is not in the buffer cache, a physical I/O request is queued. Then the query waits and the page is fetched from the disk.

Changes made to data on a page from a DELETE or UPDATE operation is also made to pages in the buffer cache. These changes are later flushed out to the disk. This whole mechanism allows SQL Server to optimize physical I/O in several ways:

- Multiple pages can be read and written in one I/O operation.
- Read ahead can be implemented. SQL Server may notice that for certain types of operations, it could be useful to read sequential pages—the assumption being that right after you read the page requested, you will want to read the adjacent page.

Note: Index fragmentation will hamper SQL Server's ability to perform read ahead optimization.

Evaluating buffer cache health

There are two indicators of buffer cache health:

- **MSSQL$Instance:Buffer Manager\Buffer cache hit ratio**—This is the ratio of pages found in cache to pages not found in cache (the pages that need to be read off disk). Ideally, you want this number to be as high as possible. It is possible to have a high hit ratio but still experience cache thrashing.
- **MSSQL$Instance:Buffer Manager\Page Life Expectancy**—This is the amount of time that SQL Server is keeping pages in the buffer cache before they are evicted. Microsoft says that a page life expectancy greater than five minutes is fine. If the life expectancy falls below this, it can be an indicator of memory pressure (not enough memory) or cache thrashing.

I'd like to conclude this section with an analogy; many people argue that the innovation of anti-lock brakes and other assisted technology means that braking distances should be reduced, and also that speed limits could be raised in line with this new technology. The warning value of 300 seconds (five minutes) for Page Life Expectancy incurs a similar debate in the SQL Server community: some believe that it is a hard and fast rule, while others believe that the increase in memory capacity in most servers these days means that the figure should be in the thousands rather than in the hundreds. This difference in opinion highlights to me the importance of baselines and why it is so important to have a detailed understanding of what the warning levels of every performance counter should be in your environment,

Cache thrashing

During a large table or index scan, every page in the scan must pass through the buffer cache, which means that possibly useful pages will be evicted to make room for pages that are not likely to be read more than once. This leads to high I/O since the evicted pages have to be read from disk again. This cache thrashing is usually an indication that large tables or indexes are being scanned.

To find out which tables and indexes are taking up the most space in the buffer cache, you can examine the sys.dm_os_buffer_descriptors DMV (available from SQL Server 2005). The sample query below illustrates how to access the list

If the database files are placed correctly and all object hotspots have been identified and separated onto different volumes, then it is time to take a close look at the hardware.

```
SELECT o.name, i.name, bd.*
  FROM sys.dm_os_buffer_descriptors bd
      INNER JOIN sys.allocation_units a
        ON bd.allocation_unit_id = a.allocation_unit_id
      INNER JOIN
      sys.partitions p
        ON    (a.container_id = p.hobt_id AND a.type IN (1, 3))
          OR (a.container_id = p.partition_id AND a.type = 2)
      INNER JOIN sys.objects o ON p.object_id = o.object_id
      INNER JOIN sys.indexes i
        ON p.object_id = i.object_id AND p.index_id = i.index_id
```

Running a server-side profiler trace will enable you to view your server's workload and pick out frequently executed pieces of code.

of tables or indexes that are consuming space in the buffer cache in SQL Server:

You can also use the index DMVs to find out which tables or indexes have large amounts of physical I/O.

Tip #1. Understand how indexes are used and find bad indexes.

SQL Server 2012 provides some very useful data on indexes, which you can fetch using DMOs implemented in version SQL Server 2005.

Using the sys.dm_db_index_operational_stats DMO

sys.dm_db_index_operational_stats contains information on current low-level I/O, locking, latching, and access method activity for each index. Use this DMF to answer the following questions:

- Do I have a "hot" index? Do I have an index on which there is contention? The row lock wait in ms/page lock wait in ms columns can tell us whether there have been waits on this index.
- Do I have an index that is being used inefficiently? Which indexes are currently I/O bottlenecks? The page_io_latch_wait_ms column can tell us whether there have been I/O waits while bringing index pages into the buffer cache—a good indicator that there is a scan access pattern.
- What sort of access patterns are in use? The range_scan_count and singleton_lookup_count columns can tell us what sort of access patterns are used on a particular index.

Figure 6 illustrates the output of a query that lists indexes by the total PAGE_IO_LATCH wait. This is very useful when trying to determine which indexes are involved in I/O bottlenecks.

Using the sys.dm_db_index_usage_stats DMO

sys.dm_db_index_usage_stats contains counts of different types of index operations and the time each type of operation was last performed. Use this DMV to answer the following questions:

- How are users using the indexes? The user_seeks, user_scans, user_lookups columns can tell you the types and significance of user operations against indexes.
- What is the cost of an index? The user_updates column can tell you what the level of maintenance is for an index.
- When was an index last used? The last_* columns can tell you the last time an operation occurred on an index.

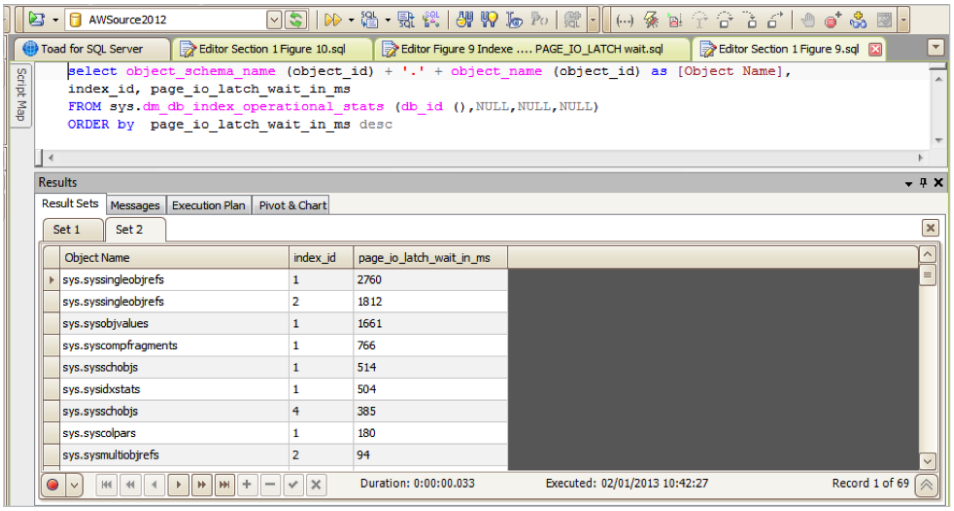

Figure 6. Indexes listed by the total PAGE_IO_LATCH wait

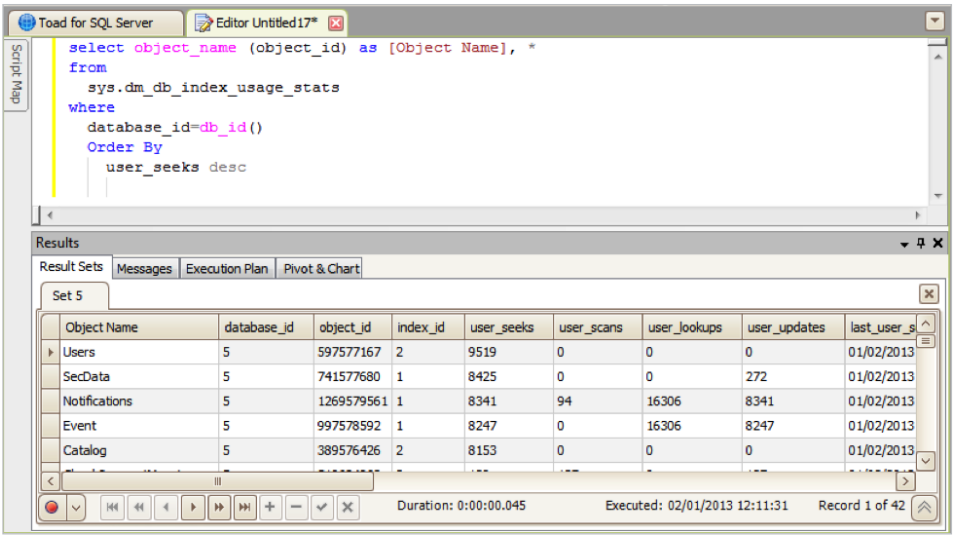

Figure 7. Indexes listed by the total number of user_seeks

Figure 7 illustrates the output of a query that lists indexes by the total number of user_seeks. If you instead wanted to identify indexes that had a high proportion of scans, you could order by the user_scans column. Now that you have an index name, wouldn't it be good if you could find out what SQL statements used that index? On SQL Server 2005 and newer versions, you can.

There are of course many other areas to indexes, such as design strategy, consolidation and maintenance. If you would like to read more about this vital area of SQL Server Performance tuning, head over to SQLServerPedia or check out some of Dell's webcasts or white papers on the subject.

Conclusion

Of course, there are far more than 10 things you should know about SQL Server performance. However, this white paper offers a good starting point and some practical tips about performance optimization that you can apply to your SQL Server environment.

To recap, remember these 10 things when optimizing SQL Server performance:

10. Benchmarking facilitates comparisons of workload behavior and lets you spot abnormal behavior because you have a good indication of what normal behavior is.
9. Performance counters give you quick and useful information about currently running operations.
8. Changing server settings can provide a more stable environment.
7. DMOs help you identify performance bottlenecks quickly.
6. Learn to use SQL Profiler,traces and Extended Events.
5. SANs are more than just black boxes that perform I/O.
4. Cursors and other bad T-SQL frequently return to haunt applications.
3. Maximize plan reuse for better SQL Server caching.
2. Learn how to read the SQL Server buffer cache and how to minimize cache thrashing.

And the number one tip for optimizing SQL Server performance:

1. Master indexing by learning how indexes are used and how to find bad indexes.

A page life expectancy of less than five minutes can indicate memory pressure (not enough memory) or cache thrashing.

Call to action

I am sure that you are eager to implement the lessons you have learned in this whitepaper. The table below lists actions to start you on the road to a more optimized SQL Server environment:

Action	Subtasks	Target date
Gain approval to start the project	Speak to your line manager and present the case that with this project in place, you can be proactive rather than reactive.	
Identify performance goals	Speak to company stakeholders to determine acceptable levels of performance.	
Establish a baseline for system performance	Collect relevant data and store it in a custom-built or third-party repository.	
Identify top performance counters and configure trace and/or extended events	Download Dell's Perfmon counter poster.	
Review server settings	Pay specific attention to the memory and "Optimize for ad hoc workloads" settings.	
Review I/O subsystem	If appropriate, speak to your SAN administrators and consider performing I/O load tests using tools such as SQLIO, or simply determine the rate at which you can do intensive read and write operations, such as when you're performing backups.	
Identify poorly performing queries	Analyze the data returned from traces, extended event sessions and the plan cache.	
Refactor poorly performing code	Check out the latest best practices from SQLServerPedia's syndicated blog service.	
Index maintenance	Ensure your indexes are as optimal as possible.	

Using parameterized queries not only improves plan reuse and compilation overhead, but it also reduces the SQL injection attack risk involved with passing parameters via string concatenation.

For More Information

About Dell

Dell Inc. (NASDAQ: DELL) listens to customers and delivers worldwide innovative technology, business solutions and services they trust and value. For more information, visit www.dell.com.

If you have any questions regarding your potential use of this material, contact:

Dell Software
5 Polaris Way
Aliso Viejo, CA 92656
www.dell.com
Refer to our Web site for regional and international office information.