

## How to Perform a SQL Server Performance Audit

### The SQL Server Performance Audit

The goal of this performance audit checklist is to help you, in a quasi-scientific way, is to help you identify any obvious performance problems with your SQL Server. As I mentioned above, SQL Server performance tuning can be difficult. What I am trying to do with this checklist is to identify all of the "easy" SQL Server performance problems, leaving the hard ones for a later time. I am doing this because it is easy to confuse the easy and the hard SQL Server performance tuning decisions. By creating a list of the "easy" performance tuning areas, it is easier to focus on getting the easy ones out of the way, and once they are out of the way, then you can focus on the more difficult ones.

One of the advantages of using this checklist to perform a SQL Server performance audit is that it will not only tell you what you can do to easily boost current performance, it also can be used to help you know what you have already done correctly. In some cases, the choices you have made for your SQL Server may be deliberately different than the recommendations found on this checklist. In other words, you have purposely made a specific decision not to follow common SQL Server performance tuning practices. In some cases, you may be right. Not all SQL Server performance recommendations are applicable to all situations. In other cases, you may have made a decision based on resource limitations, such as not having the money to purchase the necessary hardware to carry your load. If that's the case, then you have no choice but to live with this. And in other cases, the decisions you have made may be due to political reasons, which may or may not be able to be changed. In any event, you need to do what you can, using this performance audit to identify those areas that you can change, and then making those changes to boost your SQL Server's performance.

Ideally, you should perform this audit on each of your SQL Servers. If you have many of them, this could take some time. I would suggest that you start on the servers that are currently producing the most performance problems, and then working your way to the rest of the servers as you have time.

Once you complete your performance audit, you still aren't done. Remember, these are the easy ones. Once you have the easy ones out of the way, then you can begin to devote your time to working on the harder performance issues. And that's another article series for another time.

### How to Conduct Your SQL Server Performance Audit

To make your SQL Server Performance Audit easy to perform, I have divided it into several sections. They include:

- [Using Performance Monitor to Identify SQL Server Hardware Bottlenecks](#)
- [Server Hardware Performance Checklist](#)
- [Operating System Performance Checklist](#)
- [SQL Server 2000 Configuration Performance Checklist](#)
- [Database Configuration Settings Performance Checklist](#)
- [Index Performance Checklist](#)
- [Application and Transact-SQL Performance Checklist](#)
- [SQL Server Database Job Performance Checklist](#)
- [Using Profiler to Identify Poorest Performing Queries](#)
- [How to Best Implement a SQL Server Performance Audit](#)

The best way to conduct your SQL Server Performance Audit is to first review each of the above sections, and to print them out. From there, complete each section, writing down your results as you collect them. You may perform the audit in any order you like. Once you have completed your audit, you will be ready to decide what approach you want to take to implement the various recommendations discovered from taking the audit. You will learn more about this in the last section.

## Using Performance Monitor to Identify SQL Server Hardware Bottlenecks

### Performance Audit Checklist

| Counter Name                              | Average | Minimum | Maximum |
|---|---------|---------|---------|
| Memory: Pages/sec                         |         |         |         |
| Memory: Available Bytes                   |         |         |         |
| Physical Disk: % Disk time                |         |         |         |
| Physical Disk: Avg. Disk Queue Length     |         |         |         |
| Processor: % Processor Time               |         |         |         |
| System: Processor Queue Length            |         |         |         |
| SQL Server Buffer: Buffer Cache Hit Ratio |         |         |         |
| SQL Server General: User Connections      |         |         |         |

Enter your results in the table above.

### Use Performance Monitor to Help Identify SQL Server Hardware Bottlenecks

The best place to start your SQL Server performance audit is to begin with the Performance Monitor (System Monitor). By monitoring a few key counters over a 24 hour period, you should get a pretty good feel for any major hardware bottlenecks your SQL Server is experiencing.

Ideally, you should use Performance Monitor to create a log of key counters for a period of 24 hours. You will want to select a "typical" 24 hour period when it comes to deciding when to create your Performance Monitor log. For example, pick a typical business day, not a weekend or holiday.

Once you have captured 24 hours of Performance Monitor data in a log, display the recommended counters in the Graph mode of Performance Monitor, and then record the average, minimum, and maximum values in the table above. Once you have done this, then compare your results with the analysis below. By comparing your results with the recommendations below, you should be able to quickly identify any potential hardware bottlenecks your SQL Server is experiencing.

### How to Interpret Key Performance Monitor Counters

Below is a discussions of the various key Performance Monitor counters, their recommended values, and some options for helping to identify and resolve the hardware bottlenecks. Note that I have limited the number of Performance Monitor counters to watch. I have done so because our goal in this article is to find the easy and obvious performance problems. Many other [Performance Monitor counters](#) can be found discussed elsewhere on this website.

#### Memory: Pages/sec

This counter measures the number of pages per second that are paged out of RAM to disk, or paged into RAM from disk. The more paging that occurs, the more I/O overhead your server experiences, which in turn can decrease the performance of SQL Server. Your goal is to try to keep paging to a minimum, not to eliminate it.

Assuming that SQL Server is the only major application running on your server, then this figure should ideally average between zero and 20. You will most likely see spikes much greater than 20, which is normal. The key here is keeping the average pages per second less than 20.

If your server is averaging more than 20 pages per second, one of the more likely causes of this is a memory bottleneck due to a lack of needed RAM. Generally speaking, the more RAM a server has, the less paging it has to perform.

In most cases, on a physical server dedicated to SQL Server with an adequate amount of RAM, paging will average less than 20. An adequate amount of RAM for SQL Server is a server that has a Buffer Hit Cache Ratio (described in more detail later) of 99% and higher. If you have a SQL Server that has a Buffer Hit Cache Ratio of 99% or higher for a period of 24 hours, but you are getting an average paging level of over 20 during this same time period, this may indicate that you are running other applications on the physical server other than SQL Server. If this is the case, you should ideally remove those applications, allowing SQL Server to be the only major application on the physical server.

If your SQL Server is not running any other applications, and paging exceeds 20 on average for a 24 hour period, this may mean that you have changed the SQL Server memory settings. SQL Server should be configured so that it is set to the "Dynamically configure SQL Server memory" option, and the "Maximum Memory" setting should be set to the highest

level. For optimum performance, SQL Server should be allowed to take as much RAM as it wants for its own use without having to compete for RAM with other applications.

### **Memory: Available Bytes**

Another way to check to see if your SQL Server has enough physical RAM is to check the Memory Object: Available Bytes counter. This value should be greater than 5MB. If not, then your SQL Server needs more physical RAM. On a server dedicated to SQL Server, SQL Server attempts to maintain from 4-10MB of free physical memory. The remaining physical RAM is used by the operating system and SQL Server. When the amount of available bytes is near 5MB, or lower, most likely SQL Server is experiencing a performance hit due to lack of memory. When this happens, you either need to increase the amount of physical RAM in the server, reduce the load on the server, or change your SQL Server's memory configuration settings appropriately.

### **Physical Disk: % Disk Time**

This counter measures how busy a physical array is (not a logical partition or individual disks in an array). It provides a good relative measure of how busy your arrays are.

As a rule of thumb, the % Disk Time counter should run less than 55%. If this counter exceeds 55% for continuous periods (over 10 minutes or so during your 24 hour monitoring period), then your SQL Server may be experiencing an I/O bottleneck. If you see this behavior only occasionally in your 24 hour monitoring period, I wouldn't worry too much, but if it happens often (say, several times an hour), then I would start looking into finding ways to increase the I/O performance on the server, or to reduce the load on the server. Some ways to boost disk I/O include adding drives to an array (if you can), getting faster drives, adding cache memory to the controller card (if you can), using a different version of RAID, or getting a faster controller.

Before using this counter under NT 4.0, be sure to manually turn it on by going to the NT Command Prompt and entering the following: "diskperf -y", and then rebooting your server. This is required to turn on the disk counters on for the first time under Windows NT 4.0. If you are running Windows 2000, this counter is turned on by default.

### **Physical Disk: Avg. Disk Queue Length**

Besides watching the Physical Disk: % Disk Time counter, you will also want to watch the Avg. Disk Queue Length counter as well. If it exceeds 2 for continuous periods (over 10 minutes or so during your 24 hour monitoring period) for each disk drive in an array, then you may have an I/O bottleneck for that array. Like the Physical Disk: % Disk Time counter, if this happens occasionally in your 24 hour monitoring period, I wouldn't worry too much, but if it happens often, then I would start looking into finding ways to increase the I/O performance on the server, as described previously.

You will need to calculate this figure because Performance Monitor does not know how many physical drives are in your array. For example, if you have an array of 6 physical disks, and the Avg. Disk Queue Length is 10 for a particular array, then the actual Avg. Disk Queue Length for each drive is 1.66 ( $10/6=1.66$ ), which is well within the recommended 2 per physical disk.

Before using this counter under NT 4.0, be sure to manually turn it on by going to the NT Command Prompt and entering the following: "diskperf -y", and then rebooting your server. This is required to turn on the disk counters on for the first time under Windows NT 4.0. If you are running Windows 2000, this counter is turned on by default.

Use both the % Disk Time and the Avg. Disk Queue Length counters together to help you decide if your server is experiencing an I/O bottleneck. For example, if you see many time periods where the % Disk Time is over 55% and when the Avg. Disk Queue Length counter is over 2 per physical disk, you can be confident the server is having a I/O bottleneck.

### **Processor: % Processor Time**

The Processor Object: % Processor Time counter, is available for each CPU (instance), and measures the utilization of each individual CPU. This same counter is also available for all of the CPUs (total). This is the key counter to watch for CPU utilization. If the % Total Processor Time (total) counter exceeds 80% for continuous periods (over 10 minutes or so during your 24 hour monitoring period), then you may have a CPU bottleneck. If these busy periods are only occur occasionally, and you think you can live with them, that's OK. But if they occur often, you may want to consider reducing the load on the server, getting faster CPUs, getting more CPUs, or getting CPUs that have a larger on-board L2 cache.

### **System: Processor Queue Length**

Along with the Processor: % Processor Time counter, you will also want to monitor the Processor Queue Length counter. If it exceeds 2 per CPU for continuous periods (over 10 minutes or so during your 24 hour monitoring period), then you probably have a CPU bottleneck. For example, if you have 4 CPUs in your server, the Processor Queue Length should not

exceed a total of 8 for the entire server.

If the Processor Queue Length regularly exceeds the recommended maximum, but the CPU utilization is not correspondingly as high (which is typical), then consider reducing the SQL Server "max worker threads" configuration setting. It is possible the reason that the Processor Queue Length is high is because there are an excess number of worker threads waiting to take their turn. By reducing the number of "maximum worker threads", what you are doing is forcing thread pooling to kick in (if it hasn't already), or to take greater advantage of thread pooling.

Use both the Processor Queue Length and the % Total Process Time counters together to determine if you have a CPU bottleneck. If both indicators are exceeding their recommended amounts during the same continuous time periods, you can be assured there is a CPU bottleneck.

### **SQL Server Buffer: Buffer Cache Hit Ratio**

This SQL Server Buffer: Buffer Cache Hit Ratio counter indicates how often SQL Server goes to the buffer, not the hard disk, to get data. In OLTP applications, this ratio should exceed 90%, and ideally be over 99%. If your buffer cache hit ratio is lower than 90%, you need to go out and buy more RAM today. If the ratio is between 90% and 99%, then you should seriously consider purchasing more RAM, as the closer you get to 99%, the faster your SQL Server will perform. In some cases, if your database is very large, you may not be able to get close to 99%, even if you put the maximum amount of RAM in your server. All you can do is add as much as you can, and then live with the consequences.

In OLAP applications, the ratio can be much less because of the nature of how OLAP works. In any case, more RAM should increase the performance of SQL Server.

### **SQL Server General: User Connections**

Since the number of users using SQL Server affects its performance, you may want to keep an eye on the SQL Server General Statistics Object: User Connections counter. This shows the number of user connections, not the number of users, that currently are connected to SQL Server.

If this counter exceeds 255, then you may want to boost the SQL Server configuration setting, "Maximum Worker Threads" to a figure higher than the default setting of 255. If the number of connections exceeds the number of available worker threads, then SQL Server will begin to share worker threads, which can hurt performance. The setting for "Maximum Worker Threads" should be higher than the maximum number of user connections your server ever reaches.

## Tips for Using Performance Monitor CPU Counters

As you measure CPU activity with CPU counters, keep in mind that the following processes are those that use the most CPU resources in SQL Server:

- **Context switching:** This occurs when threads are switched between CPUs in your SQL Server, and excessive context switching eats up CPU resources. In some cases, context switching can be reduced by turning on Lightweight Pooling. See this website for more details on this option.
- **Recompilation:** Excessive SP recompilation wastes CPU resources and slows SQL Server. See this website for ways to reduce SP recompilation.
- **Sorting:** At some point, data may have to be sorted. But to reduce the load sorting can have on SQL Server, only sort the minimum amount of data that needs to be sorted, or consider pushing off the sorting function onto the client.
- **Hashing:** Hashing often occurs when JOINS are used. They are also used in UNIONS. Often, but not always, a HASH JOIN can be modified so that it runs as a LOOP JOIN instead, which reduces CPU activity and speeds performance. See this website for more details on LOOP JOINS.

If you want to reduce CPU activity, you will need to reduce one or more of the above SQL Server functions.

\*\*\*\*\*

Measuring the CPU activity of your SQL Server is a key way to identify potential CPU bottlenecks. The **Process Object: % Processor Time** counter is available for each CPU (instance) in your server, and measures the utilization of each individual CPU. If you have more than a single CPU, or if you have a single hyperthreading CPU, then watching this counter for specific CPUs is not of much value. Instead, monitor the **\_Total** instance, which provides you with the total overall CPU utilization for your server, which is a good overall indicator of how busy your server is.

As an alternative to the **\_Total** instance, you can also use the **System Object: % Total Processor Time** counter, which measures the average of all the CPUs in your server.

Generally speaking, if the total CPU utilization of your server exceeds 80% for continuous periods (over 10 minutes or so), then you may have a CPU bottleneck. Some potential solutions to a CPU bottleneck are to reduce the server load (tune those queries), get faster CPUs, or get more CPUs.

\*\*\*\*\*

Another valuable indicator of CPU performance is the System Object: **Processor Queue Length**. If the Processor Queue Length exceeds 2 per CPU for continuous periods (over 10 minutes or so), then you probably have a CPU bottleneck. For example, if you have 4 CPUs in your server, the Processor Queue Length should not exceed a total of 8 for the entire server.

If the Processor Queue Length regularly exceeds the recommended maximum, but the CPU utilization is not correspondingly as high (which is typical), then consider reducing the SQL Server "max worker threads" configuration setting. It is possible the reason that the Processor Queue Length is high is because there are an excess number of worker threads waiting to take their turn. By reducing the number of "maximum worker threads", what you are doing is forcing thread pooling to kick in (if it hasn't already), or to take greater advantage of thread pooling.

Use both the Processor Queue Length and the **% Total Process Time** counters together to determine if you have a CPU bottleneck. If both indicators are exceeding their recommended amounts during the same continuous time periods, you can be assured there is a CPU bottleneck.

\*\*\*\*\*

If the System Object: **% Total Processor Time** counter in your multiple CPU server regularly runs over 80% or so, then you may want to start monitoring the **System: Context Switches/Sec** counter. This counter measures how often NT Server switches between threads. Heavy context switching hurts performance and should be minimized. If the System: Context Switches/Sec counter nears 8000, then you should consider using **NT Windows Fibers**. Fibers are subcomponents of threads that perform similarly to threads. The advantage of using them is that they have less overhead when being switched between CPUs in a multiple CPU server.

This benefit does not show up unless the server's CPUs are running at near maximum, or the Performance Monitor System: Context Switching/Sec counter nears 8000 switches per second for continual periods (over 10 minutes), so don't select this option unless your server's CPUs are nearly maxed out. You will also want to carefully test (before and after) the affect of this setting on your server's performance, because you may not always get the results you expect when making this change.

\*\*\*\*\*

Sometimes, a performance monitor counter measures potential bottlenecks you may not think about. For example, there is a Performance Monitor counter called **System: % Total Privileged Time**. What this counter measures is what percent of the System: % Total Processor Time counter is used for running Kernel Mode (sometimes also referred to as privileged mode) code. As you may know, a CPU running under Windows NT, Windows 2000, or Windows 2003 can run in two modes: kernel or user mode. Most, but not all, of the operating system code is run in kernel mode, and some operating system and all user applications are run in user mode.

So what does all this mean? If you notice that the System: % Total Privileged Time counter is running at greater than 20%, this is an indication that your server's I/O may be bottlenecked. Why? Because the I/O driver runs in kernel mode, and one indication of high I/O use is indicated by a higher percentage of Privileged Time use. If this number is high, and the % Disk Time counter is over 55%, you can be fairly sure that I/O is a bottle neck in your server.

Besides intensive I/O use, a high value for this counter can indicate potential network driver, disk driver, or hardware problems. If you are getting a 25% or higher reading for the % Total Privileged Time counter, but the % Disk Time counter is less than 55%, then most likely it is not an I/O bottleneck, but a driver or hardware problem, and you should starting taking a look at these.

While this is not a counter I watch regularly, you might consider using it to help confirm your suspicions of a potential I/O bottleneck on your server.

## SQL Server Hardware Performance Checklist

### Performance Audit Checklist

| SQL Server Hardware Characteristics                | Describe Here |
|--|---------------|
| Number of CPUs                                     |               |
| CPU MHz  |               |
| CPU L2 Cache Size                                  |               |
| Physical RAM Amount                                |               |
| Total Amount of Available Drive Space on Server    |               |
| Total Number of Physical Drives in Each Array      |               |
| RAID Level of Array Used for SQL Server Databases  |               |
| Hardware vs. Software RAID                         |               |
| Disk Fragmentation Level                           |               |
| Location of Operating System                       |               |
| Location of SQL Server Executables                 |               |
| Location of Swap File                              |               |
| Location of tempdb Database                        |               |
| Location of System Databases                       |               |
| Location of User Databases                         |               |
| Location of Log Files                              |               |
| Number of Disk Controllers in Server               |               |
| Type of Disk Controllers in Server                 |               |
| Size of Cache in Disk Controllers in Server        |               |
| Is Write Back Cache in Disk Controller On or Off?  |               |
| Speed of Disk Drives                               |               |
| How Many Network Cards Are in Server?              |               |
| What is the Speed of the Network Cards in Server?  |               |
| Are the Network Cards Hard-Coded for Speed/Duplex? |               |
| Are the Network Cards Attached to a Switch?        |               |
| Are All the Hardware Drivers Up-to-Date?           |               |
| Is this Physical Server Dedicated to SQL Server?   |               |

*Enter your results in the table above.*

### Auditing SQL Server Hardware Is An Important Early Step

From the previous section, on using Performance Monitor, you may have identified some potential hardware bottlenecks that are negatively affecting your SQL Server's performance. In this section, we will take a look at each of the major components of a SQL Server's hardware, and examine what can be done to help maximize the performance of your hardware.

This portion of the audit will be divided into these major sections:

- CPU
- Memory
- Disk Storage
- Network Connectivity
- Misc.

As part of this audit, you will want to complete the above checklist. As you do, you may find out things about server you

were not aware of.

## **CPU**

### *Number of CPUs*

This first point is obvious, the more CPUs your SQL Server has, the faster it can perform. The standard edition of SQL Server 2000 can support up to 4 CPUs. The Enterprise version can support up to 32 CPUs, depending on the OS used. Multiple CPUs can be effectively used by SQL Server to boost overall performance.

It is very difficult to estimate the number of CPUs any specific SQL Server-based application will need. This is because each application works differently and is used differently. Experienced DBAs often have a feel for what kind of CPU power an application might need, although until you really test your server's configuration under realistic conditions, it is hard to really know what is needed.

Because of the difficulty of selecting the appropriate numbers of CPUs to purchase of a SQL Server, you might want to consider the following rules of thumb:

- Purchase a server with as many CPUs as you can afford.
- If you can't do the above, then at least purchase a server that has room to expand its total number of CPUs. Almost all SQL Servers need more power as time passes and workloads increase.

Here's some potential scenarios:

- SQL Server will be used to run a specialized accounting application that will only be used by no more than 5 users at a time, and you don't expect this to change in the next couple of years. If this is the case, a single CPU will most likely be adequate. If you expect that the number of users may increase fairly soon, then you would want to consider purchasing with a single CPU now, but with room to expand to a second one should the need arise.
- SQL Server will be used to run a specialty application written in-house. The application will not only involve OLTP, but need to support fairly heavy reporting needs. It is expected that concurrent usage will not exceed 25 users. In this case, you might want to consider a server with two CPUs, but with the ability to expand to 4 if necessary. It is hard to predict what "fairly heavy reporting needs" really mean. I have seen some fairly simple, but poorly written reports, peg out all of a server's CPUs.
- SQL Server will run an ERP package that will support between 100 - 150 concurrent users. For "heavy-duty" applications like this, ask the vendor for their hardware recommendations, as they should already have a good idea of their product's CPU needs.

I could provide many other examples, but the gist of what I am trying to get across is that it is very hard to predict exactly how many CPUs a particular SQL Server-based application will need, and that you should generally purchase a system bigger than you think you will need, because in many cases, usage demands on an application are often underestimated. It is less expensive in the long run to purchase a larger server now (with more CPUs), than to have to replace your entire server in 6-12 months because of poor estimates.

### **CPU Speed**

Like the number of CPUs, the needed speed of the CPUs you purchase is hard to estimate. Generally speaking, as with the number of CPUs your SQL Server has, purchase the fastest CPUs you can afford. It is better to purchase too large a system than too small a system.

### **CPU L2 Cache**

One of the most common questions I get is "should you purchase a less expensive CPU with a smaller L2 cache, or a more expensive XEON CPU with a larger L2 cache?" What complicates this decision is the fact that you can purchase faster chips with smaller L2 caches than you can of chips that have a large L2 cache. Here's my rule of thumb:

- If you will only be running 1 or 2 CPUs, go with the fastest CPU you can get, with L2 cache as a secondary consideration. If you have a choice of L2 cache size, always get the largest you can.
- But, if you will be running 4 or more CPUs, then you want to go with the CPUs with the largest L2 cache, even though their speed may not be as high. The reason for this is in order for SQL Server to run optimally on servers with four or more CPUs, the L2 cache has to be much larger, otherwise you will be wasting much of the power of the

additional CPUs.

### **CPU Audit Checklist**

Since this article is about an audit of your current SQL Server's CPU capability, your focus now should be on whether or not your current servers are experiencing any CPU bottlenecks. As was discussed in the Performance Monitor section of this article, you can use the Performance Monitor to help you identify hardware bottlenecks.

If you are not experiencing currently CPU bottlenecks, then you can skip to the next section on memory. But if your current server is experiencing a CPU bottleneck, and it is bad enough to cause major performance problems, then these are your options to resolving this bottleneck:

- Reduce the load on your server. This can be accomplished by reducing the number of users, by tuning queries, by tuning indexes, and by eliminating any unnecessary applications running on the server. One option is to move reporting needs from your production server to a SQL Server devoted to reporting only.
- Adding more memory, assuming that the CPU bottleneck is caused by a lack of memory in the server, which is a common problem.
- Adding additional CPUs if you have room in the current server.
- Upgrading to faster CPUs in your server, if this option is available.
- Purchasing a new server with more, and faster CPUs.

Unfortunately, none of these options to deal with CPU bottlenecks are extremely easy to implement, unless of course your company has unlimited money to spend. As a DBA in charge of a SQL Server with a CPU bottleneck, you have many difficult decisions to make, and lots of work ahead of you, especially if your only option, due to a lack of money, is to "reduce the load on your server."

### **Memory**

While server memory is discussed here after first discussing the CPU, don't think that it is not as important as your server's CPU. In fact, memory is probably the most important hardware ingredient for any SQL Server, affecting SQL Server's performance more than any other hardware.

When we are talking about memory, we are referring to physical RAM. Often, the word memory (in the Windows Server world) refers to physical RAM and virtual memory (swap file). This definition is not good for SQL Server because SQL Server is not really designed to use virtual memory, although it can if it has too.

Instead of using the operating system's combination of physical RAM and virtual memory, SQL Server prefers to stay in physical RAM as much as it can. The reason for this is speed. Data in RAM is much faster to retrieve than data on disk.

When SQL Server can't keep all of the data it manages in RAM (the SQL Server buffer cache), it accesses disk, similar to the way that the operating system manages virtual memory. But SQL Server's "caching" mechanism is more sophisticated and faster than what the operating system virtual memory can provide.

The fastest way to find out if your SQL Server has an adequate amount of RAM is to check the SQL Server: Buffer Cache Hit Ratio counter that was discussed in the previous page. If this counter is 99% or higher, then most likely you have enough physical RAM in your SQL Server. If this counter is between 90% and 99%, and if you are happy with your SQL Server's performance, then you probably have enough physical RAM in your SQL Server. But if you are not satisfied with your server's performance, then more RAM should be added.

If this counter is less than 90%, the odds are that your SQL Server's performance is unacceptable (if you are running OLAP, then less than 90% is generally OK), and you will want to add more RAM to your server.

Ideally, the amount of physical RAM in a SQL Server should exceed the size of the largest database on the server. This is not always possible, as many databases are very large. If you are sizing a new SQL Server, and assuming your budget is large enough, try to order your SQL Server with enough RAM to hold the entire size of the projected database. Assuming that your database is 4GB or less, then this isn't generally too much of a problem. But if your database is larger than (or is expected to grow larger than 4GB) then you may be unable to easily or affordably get more than 4GB of RAM. While SQL Server 2000 Enterprise Edition will support up to 64GB of RAM, there aren't too many affordable servers that support this much RAM.

Even if your entire database cannot fit into SQL Server buffer cache, SQL Server can still be very fast when it comes time to retrieve data. With a 99% buffer cache hit ratio, this means that 99% of the time the data SQL Server needs is already in cache, and performance will be very fast. For example, I manage one database that is 30GB, but the server only has 4GB of RAM. The buffer cache hit ratio for this server is always over 99.6%. What this means is that in most cases, users don't access all the data in a database at the same time--only a fraction of it--and that SQL Server has the ability to keep the most used data in cache all the time, so 99% of all requests are met quickly in this particular instance, even though the server has much less physical RAM than the size of the data in the database.

So what does all of this boil down to? If your buffer hit cache ratio is less than 90%, then seriously consider adding more RAM.

## **Disk Storage**

After memory, disk storage is often the most important factor affecting SQL Server's performance. It is also a complicated topic. In this section, I will focus on the "easiest" areas where disk storage performance can be bolstered.

### *Total Amount of Available Drive Space on Server*

While the performance effect isn't huge, it is important that all of your disk arrays have at least 20% of free space. This is because NTFS (which is the disk format I assume you are using) needs extra space to work efficiently. If the space is not available, then NTFS is not able to function at its full capacity and performance can degrade. It also leads to more disk fragmentation, which causes the server to work harder to read and write data.

Take a look at each of the physical disks in your SQL Server, checking to see if there is at least 20% or more of free space. If there isn't, then consider trying:

- Removing any unnecessary data from the disks (empty the recycle bin, remove temp files, remove setup files, etc.)
- Moving some of the data to disks with more space
- Adding more disk space

### *Total Number of Physical Drives in Each Array*

A disk array generally refers to two or more physical disk drives working together as a single unit. For example, a RAID 5 array might have 4 physical drives in it. So why is it important to know how many physical drives are in the one or more arrays in your SQL Server?

With the exception of mirrored arrays (which are two physical drives working together), the more physical drives that are in an array, the faster reads and writes are for that array.

For example, let's say that I want to purchase a new SQL Server with a RAID 5 array and that I need at least 100MB of available space. Let's also assume that the vendor has proposed two different array configurations:

- 4 - 36GB drives (108GB available)
- 7 - 18GB drives (108GB available)

Both of these options meet our criteria of providing at least 100MB of RAID 5 disk space. But which array will provide better read and write performance? The answer is the second choice, the 7 18GB drives. Why?

Generally speaking, the more disks that are in an array, the more disk heads there are available to read and write data. SCSI drives, for example, have the ability to read and write data simultaneously. So the more physical drives that there are in an array, the faster data is read or written to the array. Each drive in the array shares part of the workload, and the more, the better. There are some limits to this, depending on the disk controller, but generally, more is better.

So what does this mean to you? After you take a look at the number of arrays you have in your SQL Server, and the number of drives in each array, is it feasible to reconfigure your current arrays to take better advantage of the principal of more is better?

For example, let's say that your current server has two disk arrays used to store user databases. Each is a RAID 5 array with 3 18GB drives each. In this case, it might be beneficial to reconfigure these two arrays into a single array of 6 18GB

drives. Not only would this provide faster I/O, but it would also recover 18GB of hard disk space.

Take a careful look at your current configuration. You may, or may not be able to do much. But if you can, you will be able to see the benefits of your change as soon as you make them.

### ***RAID Level of Array Used for SQL Server Databases***

As you probably already know, there are various different types of disk array configurations, called RAID levels. Each has their pros and cons. Here is a brief summary of the most commonly used RAID levels, and how they can be best used in your SQL Server:

#### *RAID 1*

- Ideally, the operating system and SQL Server executables, including the operating system's swap file, should be located on a RAID 1 array. Some people locate the swap file on its own RAID 1 array, but I doubt that this really offers much of a performance boost because paging, on a well-configured server, is not much of an issue.
- If your SQL Server database(s) are very small, and all the databases can fit on a single disk drive, consider RAID 1 for the storing of all your SQL Server data files.
- Ideally, each separate transaction log should be located on its own RAID 1 array. This is because transactions logs are written to and read from sequentially, and by isolating them to their own array, sequential disk I/O won't be mixed with slower random disk I/O, and performance is boosted.

#### *RAID 5*

- Although this is the most popular type of RAID storage, it is also not the best option for optimum SQL Server I/O performance. If a database experiences more than 10% writes, and most OLTP databases do, write performance will suffer, hurting the overall I/O performance of SQL Server. RAID 5 is best used for read-only or mostly read-only databases. Testing at Microsoft has found that RAID 5 can be as much as 50% slower than using RAID 10.

#### *RAID 10*

- RAID 10 offers the best performance for SQL Server databases, although it is the most expensive RAID option. The more write intensive the database, the more important it is to use RAID 10.
- RAID 10 arrays are also a good option for transaction logs, assuming they are dedicated to a single transaction log.

Most likely, your current SQL Server configuration does not match the recommendations above. In some cases, you may be able to modify your current array configuration to come closer to what is recommended above, but in most cases, you will probably have to live with what you have until you get a new budget for a new server and array.

If you can only do one of the above recommendations, I would recommend that you move to RAID 10 over the other options. This option, above all others listed above, will give you the greatest overall boost in SQL Server I/O performance.

### **Hardware vs. Software RAID**

RAID can be implemented through hardware or software (via the operating system). There is no debate on this topic, don't ever use software RAID, it is very slow. Always use hardware RAID.

### **Disk Fragmentation Level**

If you create a new database on a brand new disk array, the database file and transaction log file created will be one contiguous file. But if your database or transaction log grows in size (and what database and transaction log doesn't), it is possible for the files to become fragmented over time. File fragmentation, which scatters pieces of your files all over a disk array, causes your disk array to work harder to read or write data, hurting disk I/O performance.

As part of your performance audit, you need to find out how defragmented your SQL Server database and transaction logs are. If you have Windows 2000 or 2003, you can use the built-in defragmentation utility to run a fragmentation analysis to see how badly the files are fragmented. If you are running Windows NT Server 4.0, then you will have to use a third party utility, such as [Diskkeeper](#) from Executive Software, to perform the analysis.

If the analysis recommends that you defragment, you should. Unfortunately, defragmenting a SQL Server's database and transaction log files is not always an easy task. Open files, such as those database and transaction log files found on a running SQL Server, cannot always be defragmented. For example, the built-in defragmentation utility cannot defrag SQL Server MDF and LDF files, but Diskkeeper 8.0 can in many cases, but not all. This means, than in some cases, you may have to bring SQL Server offline in order to defrag MDF and LDF files. And depending on how fragmented the files are, and the size of the files, this could take many hours.

But do you really have much choice about defragmenting your SQL Server files? If your I/O performance is currently adequate, then you shouldn't bother defragmenting. But if your I/O performance is a bottleneck, then defragmenting is one inexpensive way of boosting performance, albeit a time consuming one in many cases.

Ideally, you should periodically defragment your SQL Server database and transaction log files. This way, you can ensure that you don't experience any I/O performance issues because of this very common problem.

### **Location of the Operating System**

For best performance, operating system files should be on a disk array that does not include the SQL Server data files (MDBs or LDFs). In addition, they should be located on a disk array that supports either RAID 1, 5, or 10.

Generally, I install, as most people do, the operating system on drive C: of the server. I usually configure drive C: as a RAID 1 mirrored drive for both fault tolerance and best overall performance.

In most cases, as long as you don't locate the operating system on the same array as SQL Server data files, you have great flexibility in placing operating system files on your server.

### **Location of SQL Server Executables**

The location of the SQL Server executables (binaries), like the location of the operating system files, are not critical, as long as they are not located on the same array as the SQL Server data files. As with operating system files, I generally place SQL Server executables on drive C:, which is generally configured as a RAID 1 mirrored drive.

If you are building a SQL Server 7.0 cluster, then the SQL Server executables cannot be located on drive C:, but instead must be located on a shared array. Unfortunately, this is often the same array that you store the SQL Server data files, unless you have a lot of money to spend on a separate array just for the executables. While performance is somewhat hindered by locating the executables on the same shared array as the data files, it is not too bad a compromise, given the fault tolerance you are getting in return. On the other hand, this is a good reason to upgrade to SQL Server 2000 clustering. If you are building a SQL Server 2000 cluster, then the SQL Server executables have to be located on local drives, not the shared array, so performance is not an issue.

### **Location of Swap File**

Assuming that your SQL Server is a dedicated SQL Server, and that SQL Server memory usage has been set to dynamic (the default), the swap file won't see a lot of activity. This is because SQL Server doesn't normally use it a lot. Because of this, it is not critical that the swap file be located in any particular location, except you don't want to locate it on the same array as SQL Server data files.

Generally, I place the swap file on the same array as the operating system and SQL Server executables, which I have indicated earlier, is a disk array that supports RAID 1, RAID 5, or RAID 10. This is usually drive C:. This makes administration much easier.

If your SQL Server is a shared server, running applications other than SQL Server, and paging is an issue (due to the other applications), you might want to consider moving the swap file to its own dedicated array for better performance. But better yet, make SQL Server a dedicated server.

### **Location of the tempdb Database**

If your tempdb database is heavily used, consider moving it to an array of its own, either RAID 1 or RAID 10, to boost disk I/O performance. Avoid RAID 5 arrays as they can be slow when writing data, a common side-effect of using tempdb. If you can't locate the tempdb on its own array, and you want to avoid locating it on the same array as your database files, consider locating it on the same drive as the operating system. This will help to reduce overall I/O contention and boost performance.

If your application uses the tempdb database a lot, and causes it to grow larger than its default size, you may want to permanently increase the default size of the tempdb file to a size closer to what is actually used by your application on a

day-to-day basis. This is because every time the SQL Server service (mssqlserver) is restarted, the tempdb file is recreated to the default size. While the tempdb file can grow, it does take some resources to perform this task. By having the tempdb file at the correct size when SQL Server is restarted, you don't have to worry about the overhead of it growing during production.

In addition, heavy activity in the tempdb database can drag down your application's performance. This is especially true if you create one or more large temp tables and then are querying or joining them. To help speed these queries, be sure the AUTOSTATS database option is turned on for tempdb, and then create one or more indexes on these temp tables that can be used by your query. In many cases, you will find that this can substantially speed up your application. But like many performance tips, be sure you test this one to see if it actually helps in your particular situation.

## **Location of System Databases**

The system databases (master, msdb, model) don't experience a lot of read and write activity, so locating them on the same array as your SQL Server data files is generally not a performance issue. The only exception might be for very large databases with hundreds or thousands of users. In this case, putting them on their own array can help boost overall I/O performance somewhat.

## **Location of User Databases**

For best performance, user database files (MDBs) should be located on their own array (RAID 1, 5, or 10), separate from all other data files, including log files. If you have multiple large databases on the same SQL Server, consider locating each separate database file(s) on its own array for less I/O contention.

## **Location of Log Files**

Ideally, each log file should reside on its own separate array (RAID 1 or 10, RAID 5 will slow down transaction log writes more than you would like). The reason for this is because most of the time, transaction logs experience sequential writes, and if the array can write the data sequentially (not having to interrupt itself to perform other reads and writes), then sequential writes are very fast. But if the array can't write sequentially because it has to random perform other reads and writes, sequential writes can't be performed, and performance suffers.

Of course, having a separate array for each log file is expensive, and often can't be cost justified. At the very least though, locate all log files on an array (RAID 1 or RAID 10) other than the array used for database files. While sequential write performance won't be as good as if each log file had its own array, it is still much better than trying to contend for disk I/O with data files.

## **Number of Disk Controllers in Server**

A single disk controller, whether it is SCSI or fibre, has a maximum limit on its throughput. Because of this, you will want to match the number of disk controllers to the amount of data throughput you expect. As each controller is different, I can't recommend specific solutions, other than to say that at a very minimum, you will want two disk controllers. One controller should be used for non-hard disk devices, such as the CD-ROM, backup devices, and so on. And the other controller would be used for hard disk. The goal is not to attach both slow and fast devices on the same controller.

Quite often, you see this scenario, which is a good one. One controller is for non-hard disk devices, one controller is used for a RAID 1 local hard disk, and a third (and sometimes more) is used for arrays that hold SQL Server database files and logs. Be sure you don't attach more drives to a controller than it can handle. While it may work, performance will suffer.

## **Type of Disk Controllers in Server**

Always purchase the fastest disk controller you can afford, assuming you want the best SQL Server performance. As you may know, different disk controllers have different performance characteristics. For example, there are different types of SCSI, such as Wide SCSI, Narrow SCSI, Ultra SCSI, and so on. The same is true, although to a less degree, of fibre connections.

Because of the wide variety of controllers, I can't recommend any specific ones. Generally, a hardware vendor will offer several models to choose from. Ask about the performance benefits of each one, and get the one that offers the best throughput.

## **Size of Cache in Disk Controllers in Server**

Also, when you purchase a disk controller, consider how much disk cache it has. Some disk controllers allow you to add

extra disk cache. Generally, you will want to purchase as much disk cache as your controller can hold. SQL Server is very I/O intensive, and anything we can do to boost I/O performance, like employing a large disk cache, will help out a lot.

### **Is Write Back Cache in Disk Controller On or Off?**

Disk cache in your disk controller offers two ways to speed access. One is for reads, and the other for writes. Of these, the most important use is for reads, as this is where most disk I/O time is spent in most SQL Server databases. A write back cache, on the other hand, is used to speed up writes, which usually occur less often, relatively speaking. Unfortunately, SQL Server, in most cases, assumes that write back cache is not on, and because of this, write back caching should be turned off on most controllers. If you don't, it is possible, under certain circumstances, to get corrupted data after SQL Server writes data (once it writes data, it assumes it was written correctly), but for some reason (such as a loss of power) the write back cache does not write the data to disk.

While there are some controllers that offer battery backup to help prevent such issues, they don't always work as expected. Personally, I prefer non-corrupt data (written more slowly) than corrupt data (that was written much faster). In other words, I recommend turning write back caching off on your disk controller, even though you might suffer a very small write performance hit by doing so.

### **Speed of Disk Drives**

The disk drives that come in your arrays can often be purchased with different speeds. As you might expect, for best performance, always purchase the fastest disks you can. Generally, this is 15,000 RPM or faster. In addition, don't mix and match drives of different speeds in the same array. If you do, performance will suffer.

### **How Many Network Cards Are in Your Server?**

Fortunately, network traffic to and from a SQL Server is generally not a bottleneck, and a single network card is often more than adequate. But if you find that network traffic is a problem (you have hundreds or thousands of users) then moving to multiple network cards is justified, and can boost performance. In addition, two or more network cards can add to redundancy, helping to reduce downtime.

### **What is the Speed of the Network Cards in Server?**

At the very minimum, your server should have 100Mbps network cards. Ten megabit cards just don't offer the bandwidth you need. If one or more 100Mbps cards don't offer enough throughput, then consider gigabit cards. In fact, you might want to skip 100Mbps cards altogether and only use gigabit cards instead. Using a faster network card doesn't speed up network traffic, it only allows more traffic to get through, which in turn allows your server to work at its optimum performance.

### **Are the Network Cards Hard-Coded for Speed/Duplex?**

If you have a dual 10/100 or 10/100/1000 card in a SQL Server that is supposed to auto-sense the network's speed and set itself accordingly, don't accept that it has worked correctly. It is fairly common for a network card to auto-sense incorrectly, setting a less than optimum speed or duplex setting, which can significantly hurt network performance. What you need to do is to manually set the card's speed and duplex setting, this way you know for sure that it has been set correctly.

### **Are the Network Cards Attached to a Switch?**

This may be obvious in a large data center, but for smaller organizations, a hub may still be used to connect server. If so, seriously consider replacing the hub with an appropriate switch, and configure the switch to communicate at its highest possible performance, such as 100Mbps and full duplex. Moving from a hub to a switch can make dramatic difference in network performance.

### **Are All the Hardware Drivers Up-to-Date?**

Admittedly, this is a boring topic, but it is more important than you might think. One of the biggest performance hogs (not to leave out causes of strange and unusual problems) are buggy drivers, whether they are found in disk controllers, network cards, or elsewhere. By using the latest drivers, the odds are that you will be getting a better, faster performing driver, allowing SQL Server to perform at its best.

Regularly, you should be checking to see if newer drivers are available for your hardware, and installing them when you have downtime. I have personally seen radical performance differences by changing from an old, buggy driver to a new one that has been thoroughly debugged and tuned.

## **Is this Physical Server Dedicated to SQL Server?**

I have alluded to this before, but I can't say it too often. SQL Server should run on a dedicated physical server, not shared with other application software. When you share SQL Server with other software, you force SQL Server to fight over physical resources, and you make it much more difficult to tune your server for optimum SQL Server performance. Time and time again, when I get questions about poor SQL Server performance, I find out that the culprit responsible is another application running on the same server. You just have to learn to say NO.

## Operating System Performance Checklist

### Performance Audit Checklist

| Operating System Performance Considerations   | Your Configuration |
|---|--------------------|
| Which OS version are you running?   |                    |
| Are the disk partitions formatted using NTFS 5.0?   |                    |
| Is "NTFS data file encryption and compression" turned off?                                      |                    |
| Does your server have the latest service pack?  |                    |
| Does your server have the most current, Microsoft-certified hardware drivers?                   |                    |
| Is the Windows Server configured as a stand-alone server?                                       |                    |
| Is the "Application Response" setting, set to "Optimize Performance" for "Background Services?" |                    |
| Has security auditing been turned on?   |                    |
| How large is the server's PAGEFILE.SYS swap file?   |                    |
| Have unnecessary services been turned off?  |                    |
| Have all unnecessary network protocols been turned off?   |                    |
| Is antivirus software being used?   |                    |

Enter your results in the table above.

### Configuring Your Windows Server is Easy, But Critical

The focus of this section of our performance audit is the base operating system, and how to optimize it in order to get the best performance out of SQL Server.

Like SQL Server, Windows Server is mostly self-tuning. But like SQL Server, there are things we can do to help optimize Windows' performance. And every time we help boost the performance of Windows Server, we are at the same time boosting the performance of SQL Server.

### Selecting the Best Performing OS?

While SQL Server can run under Windows NT 4.0 Server, Windows 2000 and Windows 2003, the focus here is on Windows 2003, as it is the most current version of the operating system.

If you want the best performance out of SQL Server, you will want to run it under Windows 2003 server, as it offers many performance improvements over Windows 2000 and NT 4.0. Some of these include:

- The ability to better take advantage of Intel hyper-threading CPUs.
- Up to 32 CPUs and 64GB of RAM are now supported using Intel chips, and up to 64 CPUs and 512GB of RAM are supported using Itanium chips.
- I/O path and disk I/O performance have been substantially boosted, while at the same time reducing the amount of CPU resources needed to service I/O requests.

If you have not upgraded your SQL Servers to Windows 2003 yet, do so as soon as you can. It will provide a quick and easy boost to your SQL Server's performance.

### Are the Disk Partitions Formatted Using NTFS 5.0?

If your server is new and Windows 2000 or Windows 2003 has been recently installed, then any drives that have been formatted with it have been formatted using NTFS 5.0. But, if the server is older, and previously ran Windows NT 4.0 Server, and the drives have not been reformatted since upgrading to Windows 2000 or 2003, the disks most likely were formatted using NTFS 4.0.

While there is not a lot of difference between NTFS 4.0 and 5.0, there is enough to make the upgrade worth your while. NTFS 5.0 includes some new performance enhancements, which mean fewer disk accesses to find files, and generally overall faster disk reads. Before Windows 2000 and 2003, some DBAs formatted the drives or disk arrays dedicated to log files as FAT because it had a small performance benefit over NTFS 4.0. This is no longer true under NTFS 5.0, so all disks for all SQL Server should be formatted using NTFS 5.0 for best performance.

If you currently have a production SQL Server that is using NTFS 4.0 formatted partitions under Windows 2000, it may be difficult for you to convert them to NTFS 5.0. If this is the case, I would recommend that you not worry about this, as the performance hit is not huge. But if you are upgrading from Windows NT 4.0 server to Windows 2000, you will definitely want to reformat your drives using NTFS 5.0 to take advantage of every little performance benefit you can eek out of your server.

### **Is "NTFS Data File Encryption and Compression" Turned Off?**

NTFS 5.0 under Windows 2000 supports both file encryption and compression, and by default, these two features are turned off on a newly installed Windows 2000 or 2003 server. While these features do provide some benefits under limited circumstances, they do not provide any benefits for SQL Server. In fact, using one or both of these features can greatly hurt performance.

As you know, SQL Server is very I/O intensive, and anything that increases disk I/O hurts SQL Server's performance. Both file encryption and compressions significantly increase disk I/O as data files have to be manipulated on the fly as they are used. So if either file encryption or compression is used on SQL Server files, performance will greatly suffer.

If you become the DBA of a currently existing SQL Server, and are not familiar with it, check to see if anyone mistakenly have turned on either of these functions. If so, and you turn them off, you will become a performance hero to all of the server's users.

### **Does Your Server Have the Latest Service Pack?**

Every service pack I have ever seen has one or more performance enhancements. These could be because of tuning done by Microsoft, or because some previous bug has been fixed that boosts performance.

While you may not want to rush right out and install a new service pack the day it is released from Microsoft, once it has been tested positively in the real world, you should install the service pack.

### **Does Your Server Have the Most Current, Microsoft-Certified Hardware Drivers?**

On more than one occasion, I have seen older, buggy hardware drivers cause performance problems with Windows 2000 and 2003. Most commonly, these are disk- or network-related drivers.

Periodically, you should check to see that your server has the most recent, Microsoft-certified hardware drivers. You can do this by going to the hardware vendor's website, or by using Microsoft's Update service. In some cases, you may find a new driver that is available from the vendor, but has yet to be certified by Microsoft. I recommend that you be patient and wait (assuming this is practical) for the Microsoft-certified version. While increased performance is important, software stability is even more important.

### **Is the Windows 2000 Server Configured as a Stand-Alone Server?**

A Windows 2000 or 2003 server can be configured as either a stand-alone server or as a domain controller. For best performance, SQL Server should only run on a stand-alone server. This is because a domain controller has a lot of overhead that takes away server resources from SQL Server, hurting performance.

### **Is the "Application Response" Setting, Set to "Optimize Performance" for "Background Services?"**

In Windows 2000, under the "Advanced" tab of the "System" icon in "Control Panel", click on "Performance Options," and you can configure what is called the "Application Response" setting. You can choose to optimize performance for either "Applications" or "Background Services". You should choose "Background services" for best SQL Server performance, as this tells the OS that you want to favor background applications, such as SQL Server, over foreground applications.

In Windows 2003, under the "Advanced" tab of the "System" icon in "Control Panel," click on the "Setting" button under "Performance," the click on the "Advanced" tab. Here, you can change the performance to favor either "Programs" or "Background services." You should choose "Background services" for best SQL Server performance, as this tells the OS that you want to favor background applications, such as SQL Serve, over foreground applications.

Also, here, you can change the memory allocation to favor either "Programs" or the "System cache." For best SQL Server performance, select "Programs." This tells the OS to give more memory to applications, such as SQL Server, rather than the system cache.

After making these changes, you will most likely have to reboot your server.

### **Has Security Auditing Been Turned On?**

Windows 2000 and 2003 has the ability to audit virtually any activity on a server. By default, most security auditing is turned off. For best performance, no additional auditing should be turned on, as this will increase I/O activity, competing with SQL Server for the same I/O. Of course, if you have to have auditing turned on (because some manager says so), try to limit it as much as possible in order to reduce its negative effect on performance.

### **How Large is the Server's PAGEFILE.SYS Swap File?**

Microsoft recommends that the PAGEFILE.SYS file be set to 1.5 times the amount of physical RAM. The exact amount you need depends on what additional SQL Services you may be running. For example, if you are running Full-Text Search service, Microsoft recommends that your PAGEFILE.SYS file be three times physical RAM.

Microsoft's recommendations are a good starting point, but the best way to size the PAGEFILE.SYS is to monitor how much of it is used during production using the Performance Monitor Page File Object: % Usage counter, and then resize the PAGEFILE.SYS with a minimum size just slightly larger than the amount that is actually being used (based on the Performance Monitor counter), and with a maximum size of 50MB larger than the minimum size.

The PAGEFILE.SYS setting can be viewed and changed in Windows 2000 by right clicking on "My Computer", choosing "Properties", clicking on the "Advanced" tab, clicking on "Performance Options", and clicking on the "Change" button under "Virtual Memory". If you change the virtual memory settings, you will have to reboot your server for the new settings to go into affect.

In Windows 2003, the PAGEFILE.SYS setting can be viewed and changed under the "Advanced" tab of the "System" icon in "Control Panel," click on the "Setting" button under "Performance," the click on the "Advanced" tab, and then by clicking on the "Change" button under "Virtual memory."

### **Have Unnecessary Services Been Turned Off?**

For best performance, turn off any Windows 2000 or 2003 system services that aren't needed. This conserves both RAM and CPU cycles, helping to boost the overall performance of SQL Server.

Below are some of the operating system services (not a complete list) that are generally considered non-essential and can be turned off, if they are not used. Some of these services may not be installed on your server, and others will already be set to "Disabled" or "Manual," depending on how the server was installed and configured. Some of the services set to "Manual" are designed to only started when needed, and then to turn themselves off when no longer needed.

- Alerter
- Application Management
- Clipboard
- Distributed Link Tracking Server
- Fax Service
- File Replication
- FTP Service
- Indexing Service
- Internet Connection Sharing
- Intersite Messaging

- Kerberos Key Distribution Center
- License Logging Service
- Logical Disk Manager Administrative Service
- Messenger
- Microsoft Search
- NetMeeting Remote Desktop Sharing
- Network DDE
- Network DDE DSDM
- Print Spooler Service (if you won't be printing from this server)
- QoS RSVP
- Remote Access Auto Connection Manager
- Remote Procedure Call (RPC) Locator
- Routing and Remote Access
- RunAsService
- Smart Card
- Smart Card Helper
- SMTP Service
- Telnet
- Utility Manager
- Windows Installer
- World Wide Web Service

Generally, I turn off these services (assuming they are currently on) and ensure that their "Startup Type" setting is set to "Manual." Of course, if you have a need for any of these services, you don't have to turn them off.

### **Have All Unnecessary Network Protocols Been Turned Off?**

Generally, the only network protocol you need is TCP/IP if you are running SQL Server on it. Removing unnecessary network protocols on your SQL Servers helps by reducing the load on the server and by reducing unnecessary network traffic.

### **Is Antivirus Software Being Used?**

Real-time antivirus software creates a big resource hog for SQL Server, and is not recommended on production SQL Servers, especially clusters.

If you are worried about viruses, you can do remote scanning against your SQL Servers on a daily basis, preferably during off hours.

## SQL Server Configuration Performance Checklist

### Performance Audit Checklist

| SQL Server Configuration Settings | Advanced Setting? | Requires Restart? | Default Value | Current Value |
|-----------------------------------|-------------------|-------------------|---------------|---------------|
| affinity mask                     | Yes               | Yes               | 0             |               |
| awe enabled                       | Yes               | Yes               | 0             |               |
| cost threshold for parallelism    | Yes               | No                | 5             |               |
| cursor threshold                  | Yes               | No                | -1            |               |
| fill factor (%)                   | Yes               | Yes               | 0             |               |
| index create memory (KB)          | Yes               | No                | 0             |               |
| lightweight pooling               | Yes               | Yes               | 0             |               |
| locks                             | Yes               | Yes               | 0             |               |
| max degree of parallelism         | Yes               | No                | 0             |               |
| max server memory (MB)            | Yes               | No                | 2147483647    |               |
| max text repl size (B)            | No                | No                | 65536         |               |
| max worker threads                | Yes               | Yes               | 255           |               |
| min memory per query (KB)         | Yes               | No                | 1024          |               |
| min server memory (MB)            | Yes               | No                | 0             |               |
| nested triggers                   | No                | No                | 1             |               |
| network packet size (B)           | Yes               | No                | 4096          |               |
| open objects                      | Yes               | Yes               | 0             |               |
| priority boost                    | Yes               | Yes               | 0             |               |
| query governor cost limit         | Yes               | No                | 0             |               |
| query wait (s)                    | Yes               | No                | -1            |               |
| recovery interval (m)             | Yes               | No                | 0             |               |
| scan for startup procs            | Yes               | No                | 0             |               |
| set working set size              | Yes               | Yes               | 0             |               |
| user connections                  | Yes               | Yes               | 0             |               |

Enter your results in the table above.

### Most SQL Server Configuration Settings Should Not Be Changed

In this section, we are going to take a look at some of the performance-related SQL Server configuration settings. These are SQL Server-specific settings that can be modified using either Enterprise Manager or SP\_CONFIGURE.

As the title of this section says, in most cases, you should not modify the default SQL Server configuration settings. This is because most of the default settings provided will provide the optimum performance for most SQL Servers. And most of all, if you are not exactly sure of what the implications are of changing a setting, it is possible to hurt your server's performance instead of boosting it.

If this is the first time you have dealt with this particular SQL Server, one of your first steps should be to review the various configuration settings and then compare them to default settings in order to see which ones, if any, have been changed from the defaults. Once you have identified any of the changed settings, your next goal should be to find out why they were changed. If you can't find out why, or if you do find out why, but the reasoning behind the change is flimsy, then you will want to change the settings back to the default values. Once you have done this, your next step is to review all of the other settings (those that were set to default when you started) and evaluate each one in order to see if there might be a benefit of changing the value from the default value to a more appropriate value.

The focus of this article will be SQL Server 2000, although most of the advice applies equally to SQL Server 7.0. Before trying any of these suggestions under SQL Server 7.0, you will want to review the configuration setting section in the SQL Server 7.0 Books Online just to be sure.

There are a total of 36 different SQL Server configuration settings in SQL Server 2000. We will only focus on 23 key performance-related ones here.

## Getting Started

The easiest way to begin your audit of a SQL Server's configuration settings is to run the following command, for each of your servers, in Query Analyzer:

```
SP_CONFIGURE
```

This will produce a table similar to this one:

| name                           | minimum     | maximum    | config_value | run_value  |
|--------------------------------|-------------|------------|--------------|------------|
| affinity mask                  | -2147483648 | 2147483647 | 0            | 0          |
| allow updates                  | 0           | 1          | 0            | 0          |
| awe enabled                    | 0           | 1          | 0            | 0          |
| c2 audit mode                  | 0           | 1          | 0            | 0          |
| cost threshold for parallelism | 0           | 32767      | 5            | 5          |
| cursor threshold               | -1          | 2147483647 | -1           | -1         |
| default full-text language     | 0           | 2147483647 | 1033         | 1033       |
| default language               | 0           | 9999       | 0            | 0          |
| fill factor (%)                | 0           | 100        | 0            | 0          |
| index create memory (KB)       | 704         | 2147483647 | 0            | 0          |
| lightweight pooling            | 0           | 1          | 0            | 0          |
| locks                          | 5000        | 2147483647 | 0            | 0          |
| max degree of parallelism      | 0           | 32         | 1            | 1          |
| max server memory (MB)         | 4           | 2147483647 | 2147483647   | 2147483647 |
| max text repl size (B)         | 0           | 2147483647 | 65536        | 65536      |
| max worker threads             | 32          | 32767      | 255          | 255        |
| media retention                | 0           | 365        | 0            | 0          |
| min memory per query (KB)      | 512         | 2147483647 | 1024         | 1024       |
| min server memory (MB)         | 0           | 2147483647 | 0            | 0          |
| nested triggers                | 0           | 1          | 1            | 1          |
| network packet size (B)        | 512         | 65536      | 4096         | 4096       |
| open objects                   | 0           | 2147483647 | 0            | 0          |
| priority boost                 | 0           | 1          | 0            | 0          |
| query governor cost limit      | 0           | 2147483647 | 0            | 0          |
| query wait (s)                 | -1          | 2147483647 | -1           | -1         |
| recovery interval (min)        | 0           | 32767      | 0            | 0          |
| remote access                  | 0           | 1          | 1            | 1          |
| remote login timeout (s)       | 0           | 2147483647 | 5            | 5          |
| remote proc trans              | 0           | 1          | 0            | 0          |
| remote query timeout (s)       | 0           | 2147483647 | 600          | 600        |
| scan for startup procs         | 0           | 1          | 0            | 0          |
| set working set size           | 0           | 1          | 0            | 0          |
| show advanced options          | 0           | 1          | 1            | 1          |
| two digit year cutoff          | 1753        | 9999       | 2049         | 2049       |
| user connections               | 0           | 32767      | 0            | 0          |
| user options                   | 0           | 32767      | 0            | 0          |

The first column, "name," is the name of the SQL Server configuration setting. The second column, "minimum," is the smallest legal value for the setting. The third column, "maximum," is the largest legal value for the setting. The fourth column, "config\_value," is what the setting has been set to (but may or may not be what SQL Server is actually running now. Some settings don't go into effect until SQL Server has been restarted, or until the RECONFIGURE WITH OVERRIDE option has been run, as appropriate.) And the last column, "run\_value," is the value of the setting currently in effect. If you have not changed any of these values since the last time you restarted SQL Server, then the values in the last two columns will always be the same.

Unfortunately, the default values for these settings are not listed when you run SP\_CONFIGURE. For your convenience, this article lists the default values of those configuration settings we discuss here (see chart above).

## How to Change SQL Server Configuration Settings

Most, but not all, of the SQL Server configuration settings can be changed using Enterprise Manager. But one of the easiest ways to change any of these settings is to use the SP\_CONFIGURE command, like this:

```
SP_CONFIGURE ['configuration name'], [configuration setting value]
GO
RECONFIGURE WITH OVERRIDE
GO
```

where:

*configuration name* = The name of the configuration setting (see the name in the table above). Note that the name must be enclosed in single quote marks (or double quote marks, depending on Query Analyzer's configuration)

*configuration setting value* = The numeric value of the setting (with no quote marks).

Once SP\_CONFIGURE has run, you must perform one additional step. You must either run the RECONFIGURE option (normal settings) or the RECONFIGURE WITH OVERRIDE option (used for settings that can get you into trouble if you make a mistake), otherwise your setting change will not go into effect. Rather than trying to remember when to use each different version of the RECONFIGURE command, it is easier to just use RECONFIGURE WITH OVERRIDE all the time, as it works with all configuration settings. If you use Enterprise Manager to change a setting, it will execute RECONFIGURE WITH OVERRIDE automatically, so you don't have to.

Once you do this, most, but not all, settings go into effect immediately. For those that don't go into effect after RECONFIGURE, the SQL Server service has to be stopped and restarted.

Before we are finished with this topic, there is one more thing you need to know. Some of the configuration settings are considered "advanced" settings. Before you can change these options using the SP\_CONFIGURE command, you must first change one of the SQL Server configuration settings to allow you to change them. The command to do this is:

```
SP_CONFIGURE 'show advanced options', 1
GO
RECONFIGURE
GO
```

Only after you have run the above code may you now run SP\_CONFIGURE to change an advanced SQL Server configuration setting.

Now that you know how to change the SQL Server configuration options, let's take a look at those that are related to performance.

## Affinity Mask

When SQL Server is run under Windows Server, a SQL Server thread can move from one CPU to another. This feature allows SQL Server to run multiple threads at the same time, generally resulting in better load balancing among the CPUs in the server. The only downside to this process is that each time a thread moves from one CPU to another, the processor cache has to be reloaded, which can hurt performance in some cases.

In cases of heavily-loaded servers with *more than* 4 CPUs, performance can be boosted by specifying (to a limited degree) which processor(s) should run a specific thread. This reduces the number of times that the processor cache has to be reloaded, helping to eek out a little more performance of the server. For example, you can specify that SQL Server will only use some of the CPUs, not all of them available to it in a server.

The default value for the "affinity mask" setting, which is "0," tells SQL Server to allow the Windows Scheduling algorithm to set a thread's affinity. In other words, the operating system, not SQL Server, determines which threads run on which CPU, and when to move a thread from one CPU to another CPU. In any server with 4 or less CPUs, the default value is the best overall setting. And for servers with more than 4 CPUs, and that are not overly busy, the default value is also the best overall setting for optimum performance.

But for servers with more than 4 CPUs, and are heavily loaded because of one or more non-SQL Server applications are running on the same server as SQL Server, then you might want to consider changing the default value for the "affinity mask" option to a more appropriate value. Please note that if SQL Server is the only application running on the server, then using the "affinity mask" to limit CPU use could hurt performance, not help it.

For example, let's say you have a server that is running SQL Server, multiple COM+ objects, and IIS. Let's also assume that the server has 8 CPUS and is very busy. By reducing the number of CPUs that can run SQL Server from 8 to 4, what will happen is that SQL Server threads will now only run on 4 CPUs, not 8 CPUs. This will reduce the number of times that

a SQL Server thread can jump CPUs, reducing how often the processor cache has to be reloaded, helping to reduce CPU overhead and potentially boosting performance somewhat. The remaining 4 CPUs will be used by the operating system to run the non-SQL Server applications, helping them also to reduce thread movement and boosting performance.

For example, if you have a 8 CPU system, the value you would use in the SP\_CONFIGURE command to select which CPUs that SQL Server should only run on are listed below:

| <b>Decimal Value</b> | <b>Allow SQL Server Threads on These Processors</b> |
|----------------------|---|
| 1                    | 0   |
| 3                    | 0 and 1   |
| 7                    | 0, 1, and 2   |
| 15                   | 0, 1, 2, and 3                                      |
| 31                   | 0, 1, 2, 3, and 4                                   |
| 63                   | 0, 1, 2, 3, 4, and 5                                |
| 127                  | 0, 1, 2, 3, 4, 5, and 6                             |

Specifying the appropriate affinity mask is not an easy job, and you should consult the SQL Server Books Online before doing so for additional information. Also, you should test what happens to your SQL Server's performance before and after you make any changes to see if the value you have selected hurts or helps performance. Other than trial and error, there is no easy way to determine the optimum affinity mask value for your particular server.

As part of your audit, if you find that an affinity mask is being used, try to find out why. If there are no good answers, remove it, and return to the default value.

### **Awe Enabled**

If you are using SQL Server 2000 Standard Edition under Windows 2000 or 2003 (any version), or are running SQL Server 2000 Enterprise Edition under Windows 2000 or 2003 Server, or if your server has less than 4GB of RAM, the "awe enabled" option should always be left to the default value of 0, which means that AWE memory is not being used.

The AWE (Advanced Windowing Extensions) API allows applications (that are written to use the AWE API) to run under Windows 2000 or 2003 Advanced Server, or Windows 2000 or 2003 Datacenter Server, to access *more than* 4GB of RAM. SQL Server 2000 Enterprise Edition (not SQL Server 2000 Standard Edition) is AWE-enabled and can take advantage of RAM in a server over 4GB. If the operating system is Windows 2000 or 2003 Advanced Server, SQL Server 2000 Enterprise Edition can use up to 8GB of RAM. If the operating system is Windows 2000 or 2003 Datacenter Server, SQL Server 2000 Enterprise can use up to 64GB of RAM.

By default, if a physical server has more than 4GB of RAM, Windows 2000 and 2003 (Advanced and Datacenter), along with SQL Server 2000 Enterprise Edition, cannot access any RAM greater than 4GB. In order for the operating system and SQL Server 2000 Enterprise Edition to take advantage of the additional RAM, two steps have to be completed.

Exactly how you configure AWE memory support depends on how much RAM your server has. Essentially, to configure Windows 2000 or 2003 (Advanced or Datacenter), you must enter one of the following switches in the boot line of the boot.ini file, and reboot the server:

- 4GB RAM: /3GB (AWE support is not used)
- 8GB RAM: /3GB /PAE
- 16GB RAM: /3GB /PAE
- 16GB + RAM: /PAE

The /3GB switch is used to tell the OS to allow SQL Server to take advantage of 3GB out of the base 4GB of RAM that Windows 2000 and 2003 supports natively. If you don't specify this option, then SQL Server will only take advantage of 2GB of the first 4GB of RAM in the server, essentially wasting 1GB of RAM.

AWE memory technology is used only for the RAM that exceeds the base 4GB of RAM, that's why the /3GB switch is

needed to use as much of the RAM in your server as possible. If your server has 16GB or less of RAM, then using the /3GB switch is important. But if your server has more than 16GB of RAM, then you must not use the /3GB switch. The reason for this is because the 1GB of additional RAM provided by adding the /3GB switch is needed by the operating system in order to take advantage of all of the extra AWE memory. In other words, the operating system needs 2GB of RAM itself to manage the AWE memory if your server has more than 16GB of RAM. If 16GB or less of RAM is in a server, then the operating system only needs 1GB of RAM, allowing the other 1GB of RAM for use by SQL Server.

Once this step is done, the next step is to set the "awe enabled" option to 1, and then restart the SQL Server service. Only at this point will SQL Server be able to use the additional RAM in the server.

One caution about using the "awe enabled" setting is that after turning it on, SQL Server no longer dynamically manages memory. Instead, it takes all of the available RAM (except about 128MB which is left for the operating system). If you want to prevent SQL Server from taking all of the RAM, you must set the "max server memory" option (described in more detail later in this article) to a figure that limits SQL Server to the amount of RAM you specify.

As part of your audit process, you will want to check what this setting is and then determine if the setting matches your server's hardware and software configuration. If not, then change the setting appropriately.

### **Cost Threshold for Parallelism**

Using parallelism to execute a SQL Server query has its costs. This is because it takes a little additional overhead to run a query in parallel than to run it serially. But if the benefits of running a query using parallelism is higher than the costs, then using parallelism is a good thing.

As a rule of thumb, if a query can run serially very fast, there is no point in even considering parallelism for the query, as the extra time required to evaluate it for possible parallelism might be longer than the time it takes to run the query serially.

By default, if the Query Optimizer determines that a query will take less than 5 seconds to execute, parallelism is not considered by SQL Server. This 5 second figure can be modified using the "cost threshold for parallelism" SQL Server option. You can change this value anywhere from 0 to 32767 seconds. So if you set this value to 10, this means that the Query Optimizer won't consider parallelism for any query that it thinks will take less than 10 seconds to run.

In most cases, you should not change this setting. But if you find that your SQL Server runs many queries with parallelism, and if the CPU rate is very high, raising this setting from 5 to a higher figure (you will have to experiment to find the ideal figure for your situation), will reduce the number of queries using parallelism, also reducing the overall usage of your server's CPUs, which may help the overall performance of your server.

Another option to consider is to reduce the value from 5 seconds to a smaller number, although this could hurt, rather than help performance in many cases. One area where a smaller value might be useful is in cases where SQL Server is acting as a data warehouse and many very complex queries are being run. A lower value will allow the Query Optimizer to use parallelism more often, which can help in some situations.

You will want to test changes to the default value thoroughly before implementing it on your production servers.

If SQL Server only has access to a single CPU (either because there is only one CPU in the server, or because of an "affinity mask" setting), parallelism is not considered for a query.

If you find in your audit that the cost threshold for parallelism is being used, find out why. If you can't get an answer, move it back to the default value.

### **Cursor Threshold**

If your SQL Server does not use cursors, or uses them very little, then this setting should never be changed from its default value of "-1".

A "cursor threshold" of "-1" tells SQL Server to execute all cursors synchronously, which is the ideal setting if the result sets of cursors executed on your server are not large. But if many, or all of the cursors running on your SQL Server produce very large result sets, then executing cursors synchronously is not the most efficient way to execute a cursor.

The "cursor threshold" setting has two other options (besides the default) for running large cursors. A setting of "0" tells SQL Server to run all cursors asynchronously, which is more efficient if most or all of the cursor's result sets are large.

What if some of the cursor result sets are small and some are large, then what do you do? In this case, you can decide what large and small is, and then use this number as the cutoff point for SQL Server. For example, let's say that we consider any cursor result set of under 1000 rows as small, and any cursor result set of over 1000 rows as large. If this is the case, we can set the "cursor threshold" to 1000.

When the "cursor threshold" is set to 1000, what happens is that if the Query Optimizer predicts that the result set will be less than 1000, then the cursor will be run synchronously. And if the Query Optimizer predicts that the result set will be more than 1000, then the cursor will be run asynchronously.

In many ways, this option provides the best of both worlds. The only problem is what is the ideal "cursor threshold". To determine this, you will need to test. But as you might expect, the default value is often the best, and you should only change this option if you know for sure that your application uses very large cursors and that you have tested this option and know for sure that by changing it, it has helped, not hurt performance.

As a part of your audit, you may also want to investigate how often cursors are used, and how large the result sets are. Only by knowing this will you know what the best setting is for your server. Of course, you could always try to eliminate the use of cursors on the server. This way, the setting can remain at the default value, and you don't have to worry about the overhead of cursors.

### **Fill Factor (%)**

This option allows you to change the default fill factor for indexes when they are built. By default, the fill factor setting is set to "0". A setting of "0" is somewhat confusing, as what it means is that leaf index pages are filled 100% (not 0%), but that intermediate index pages (non-leaf pages) have some space left in them (they are not filled up 100%). Legal settings for the fill factor setting range from 0 through 100.

The default fill factor only comes into play when you build indexes without specifying a specific fill factor. If you do specify a fill factor when you create a new index, that value is used, not the default fill factor.

In most cases, it is best to leave the default fill factor alone, and if you want a value other than the default fill factor, then specify it when you create an index.

As a part of your audit, note if the fill factor is some figure other than the the default value of "0". If it is, try to find out why. And if you can't find out why the default value was changed, or there is not a good reason, switch it back to the default value. Also, if the value has been changed, keep in mind that any indexes created after the default value was changed may be using this default fill factor value. If so, you may need to re-evaluate these indexes to see if the fill factor used for creating them is appropriate.

## SQL Server Database Settings Performance Checklist

### Performance Audit Checklist

| Database Configuration Settings | Default Value             | Current Value |
|---------------------------------|---------------------------|---------------|
| auto_close                      | off                       |               |
| auto_create_statistics          | on                        |               |
| auto_update_statistics          | on                        |               |
| auto_shrink                     | off                       |               |
| read_only                       | off                       |               |
| torn_page_detection             | on in 2000<br>off in 7.0  |               |
| compatibility level             | 80 for 2000<br>70 for 7.0 |               |
| database auto grow              | on                        |               |
| transaction log auto grow       | on                        |               |

Enter your results in the table above.

### Each Database Needs to Be Audited

As part of your performance audit, you need to examine each database located on your server and examine some basic database settings. When compared to some of our other performance audit tasks, you will find this audit one of the easiest. For convenience, you may want to consider photocopying a copy of the above chart, producing one copy for each database that you will be auditing.

As a part of our database settings audit, we will be taking a look at two different types of settings: database options and database configuration settings. As in previous sections of our performance audit, we will only focus on those database settings that are directly related to performance, ignoring the rest.

Both database options and database configuration settings can be viewed or modified using Enterprise Manager (my preference, as it is easier) or modified with the ALTER DATABASE command. In addition, for the database options only, you can also use the sp\_dboption system stored procedure to view and modify them, but Microsoft is trying to phase this command out, and discourages its use (as of SQL Server 2000).

The first section of the database settings performance checklist focuses on database options, and the second section focuses on database configuration settings.

### Viewing Database Options

In this section, we will only be taking a look at six of the many database options that in one way or another can affect performance. The best way to view the current settings is to use Enterprise Manager, following these steps (These steps assume you are using SQL Server 2000):

- In Enterprise Manager, display all of the databases for your server.
- Right-click on the database you want to examine and select "Properties."
- From the Properties dialog box, select the "Options" tab.

From this screen, you can see all of the relevant database options. Note that not every database option can be seen here, but all the ones that we are interested are all listed here. Let's take a look at the performance-related ones and see how they affect SQL Server's performance.

### Auto\_Close

This database option is designed for use with the Desktop version of SQL Server 7.0 and 2000, not for the server versions. Because of this, it should not be turned on (which it is not, by default). What this option does is to close the database

when the last database user disconnects from the database. When a connection requests access to the database after it has been closed, then the database has to be reopened, which takes time and overhead.

The problem with this is that if the database is accessed frequently, which is the most likely case, then the database may close and reopened often, which puts a large performance drag on SQL Server and the applications or users making the connection.

As part of your audit, if you find this option turned on, and you are not using the desktop version of SQL Server, then you will need to research why it was turned on. If you can't find the reason, or if the reason is poor, turn this option off.

### **Auto\_Create\_Statistics**

When `auto_create_statistics` is turned on (which it is by default), statistics are automatically created on all columns used in the `WHERE` clause of a query. This occurs when a query is optimized by the Query Optimizer for the first time, assuming the column doesn't already have statistics created for it. The addition of column statistics can greatly aid the Query Optimizer so that it can create an optimum execution plan for the query.

If this option is turned off, then missing column statistics are not automatically created, which can mean that the Query Optimizer may not be able to produce the optimum execution plan for the query, and the query's performance may suffer. You can still manually create column statistics if you like, even when this option is turned off.

There is really no down-side to using this option. The very first time that column statistics are created, there will be a short delay as they are created before the query runs for the first time, causing the query to potentially take a little longer to run. But once the column statistics have been created, each time the same query runs, it should now run more efficiently than if the statistics did not exist in the first place.

As part of your audit, if you find this option turned off, you will need to research why it was turned off. If you can't find the reason, or if the reason is poor, turn this option on.

### **Auto\_Update\_Statistics**

In order for the Query Optimizer to make smart query optimization decisions, the column and index statistics need to be up-to-date. The best way to ensure this is to leave the `auto_update_statistics` database option on (the default setting). This helps to ensure that the optimizer statistics are valid, helping to ensure that queries are properly optimized when they are run.

But this option is not a panacea. When a SQL Server database is under very heavy load, sometimes the `auto_update_statistics` feature can update the statistics on large tables at inappropriate times, such as the busiest time of the day.

If you find that the `auto_update_statistics` feature is running at inappropriate times, you may want to turn it off, and then manually update the statistics (using `UPDATE STATISTICS`) when the database is under a less heavy load.

But again, consider what will happen if you do turn off the `auto_update_statistics` feature. While turning this feature off may reduce some stress on your server by not running at inappropriate times of the day, it could also cause some of your queries not to be properly optimized, which could also put extra stress on your server during busy times.

Like many optimization issues, you will probably need to experiment to see if turning this option on or off is more effective for your environment. But as a rule of thumb, if your server is not maxed out, then leaving this option on is probably the best decision.

### **Auto\_Shrink**

Some databases need to be shrunk periodically in order to free up disk space as older data is deleted from the database. But don't be tempted to use the `auto_shrink` database option, as it can waste SQL Server resources unnecessarily.

By default, the `auto_shrink` option is turned off, which means that the only way to free up empty space in a database is to do so manually. If you turn this option on, SQL Server will then check every 30 minutes to see if it needs to shrink the database. Not only does this use up resources that could better be used elsewhere, it also can cause unexpected bottlenecks in your database when the `auto_shrink` process kicks in and does its work at the worst possible time.

If you need to shrink databases periodically, perform this step manually using the `DBCC SHRINKDATABASE` or `DBCC SHRINKFILE` commands, or you can use the SQL Server Agent or create a Database Maintenance Plan to schedule regular

file shrinking during less busy times.

As part of your audit, if you find this option turned on, you will need to research why it was turned off. If you can't find the reason, or if the reason is poor, turn this option off.

## Read\_Only

If a database will be used for read-only purposes only, such as being used for reporting, consider setting the read\_only setting on (the default setting is off). This will eliminate the overhead of locking, and in turn, potentially boost the performance of queries that are being run against it. If you need to modify the database on rare occasions, you can also turn the setting off, make your change, then turn it back on.

## Torn\_Page\_Detection

Because data pages in SQL Server (8K) and NT Server or Windows Server (512 bytes) are different sizes, it is possible during power failures, or if you have disk driver or physical disk problems, for your database to become corrupted.

Here's why. Every time the operating system writes an 8K SQL Server data page to disk, it must break up the data into multiple 512 byte pages. After the first 512 byte of data is written, SQL Server assumes that the entire 8K has been written to disk successfully. So if the power should go out before all of the 512 byte pages that make up the 8K SQL Server page are written, then SQL Server does not know what has happened. This is known as a torn page.

As you can imagine, this corrupts the data page, and in effect makes your entire database corrupt. There is no way to fix a database made corrupt due to a torn page, except by restoring a known good backup. One of the best ways to prevent this problem is to ensure your server has battery backup. But this does not prevent all problems, because a defective disk driver can also cause similar problems (I have seen this.)

If you are worried about getting torn pages in your SQL Server databases, you can have SQL Server tell you if they occur (although it can't prevent them or fix them after they have occurred). There is a database option called "torn page detection" that can be turned on and off at the database level. If this option has been turned on, and if a torn page is discovered, the database is marked as corrupt and you have little choice but to restore your database with your latest backup.

In SQL Server 7.0, this option is turned off by default, and you must turn it on for every database you want it on for. In SQL Server 2000, this option is turned on by default for all databases.

So what's the big deal, why not just turn it on and be safe? The problem is that turning this feature on hurts SQL Server's performance. Not much mind you, but if you already have a SQL Server that is maxed out, then it might make a noticeable difference, and you may want to keep this option turned off. As a DBA, you must weight the pros and cons of using this option, and make the best decision for your particular situation.

## Viewing Database Configuration Settings

In this section, we will only be taking a look at three database configuration settings, and examine how they can affect performance. The best way to view these is to use Enterprise Manager, following these steps (These steps assume you are using SQL Server 2000):

- In Enterprise Manager, display all of the databases for your server.
- Right-click on the database you want to examine and select "Properties."
- From the Properties dialog box, select the "Options" tab to see the compatibility level, select the "Data Files" tab to see the database auto grow setting, and select the "Transaction Log" tab to see the transaction log auto grow setting.

Let's take a look at each of the three relevant database configuration settings.

## Compatibility Level

SQL Server 7.0 and 2000 have a database compatibility mode that allows applications written for previous versions of SQL Server to run under SQL Server 7.0 or 2000. In you want maximum performance for your database, you don't want to run your database in compatibility mode (not all new performance-related features are supported).

Instead, your databases should be running in native SQL Server 7.0 or 2000 mode (depending on which version you are currently running). Of course, this may require you to modify your application to make it SQL Server 7.0 or 2000 compliant, but in most cases, the additional work required to update your application will be more than paid for with improved performance.

SQL Server 7.0 compatibility level is referred to as "70" and SQL Server 2000 compatibility level is referred to as "80".

### **Database and Transaction Log Auto Grow**

We will be discussing both database auto grow and transaction log auto grow together because they are so closely related.

If you set your SQL Server 7.0 or SQL 2000 databases and transaction logs to grow automatically (which is the default setting), keep in mind that every time this feature kicks in, it takes up a little extra CPU and I/O time. Ideally, we want to minimize how often automatic growth occurs in order to reduce unnecessary overhead.

One way to help do this is to size the database and transaction logs as accurately as possible to their "final" size. Sure, this is virtually impossible to get right-on-target. But the more accurate your estimates (and some times it takes some time to come up with a good estimate), the less SQL Server will have to automatically grow its database and transaction logs, helping to boost performance of your application.

This recommendation in particular is important to follow for transaction logs. This is because the more times that SQL Server has to increase the size of a transaction log, the more transaction log virtual files that have to be created and maintained by SQL Server, which increases recovery time, should your transactions log need to be restored. A transaction virtual file is used by SQL Server to internally divide and manage the physical transaction log file.

The default growth amount is 10% for databases and transaction logs. This automatic growth number may or may not be ideal for your database or transaction log. If you find that your database or log is growing automatically often (such as daily or several times a week), change the growth percentage to a larger number, such as 20% or 30%. Each time the database or log has to be increased, SQL Server will suffer a small performance hit. By increasing the amount the database grows each time, the less often it will have to grow.

If your database is very large, 10GB or larger, you may want to use a fixed growth amount instead of a percentage growth amount. This is because a percentage growth amount can be large on a large database. For example, a 10% growth rate on a 10GB database means that when the database grows, it will increase by 1GB. This may or may not be what you want. If this is more than you want, then choose a fixed growth rate, such as 100MB at a time, might be more appropriate.

As part of your audit, you will need to carefully evaluate your databases to see how the above advice applies to them, then take the appropriate action.

## SQL Server Database Index Performance Checklist

### Index Performance Audit Checklist

| Indexing Checklist   | Your Response |
|--|---------------|
| Have you run the Index Tuning Wizard recently?                                 |               |
| Does every table in each database have a clustered index?                      |               |
| Are any of the columns in any table indexed more than once?                    |               |
| Are there any indexes that are not being used in queries?                      |               |
| Are the indexes too wide?  |               |
| Are tables that are JOINed have the appropriate indexes on the JOINed columns? |               |
| Are the indexes unique enough to be useful?                                    |               |
| Are you taking advantage of covering indexes?                                  |               |
| How often are indexes rebuilt?   |               |
| What is your index fillfactor?   |               |

*Enter your results in the table above.*

### Auditing Index Use is Not an Easy Task, But Critical to Your Server's Performance

When it comes to auditing index use in SQL Server databases, I sometimes get overwhelmed. For example, how to do you go about auditing indexes in a database with over 1,500 tables? While auditing a single index is relatively straight-forward, auditing thousands of them in multiple databases is not an easy task. Whether the task is easy or not, it is an important task if you want to optimize the performance of your SQL Server databases.

There are two different ways to approach the task of auditing large numbers of indexes. One option is to break down the chore into smaller, more manageable units, first focusing on those indexes that are most likely to affect the overall performance of your SQL Server. For example, you might want to start your audit on the busiest database on your server, and if that database has many tables, first start on those tables with the most data, and then working down to other tables with less data. This way, you will focus your initial efforts in areas where it will most likely have the great positive impact on your server's performance.

Another option, and the one I generally follow (because I am somewhat lazy), is to use a more of a "management by exception" approach. What I mean by this is that if I don't see any performance problems in a database, there is not much use in evaluating every index in the database. But if a database is demonstrating performance problems, then there is a good chance that indexes are less than optimal, and that I should pay extra attention to them, especially if the databases are mission critical. And if there are a lot of indexes to audit, then I start by focusing on the largest ones first, as they are the ones most likely to cause performance problems. For example, in the case of the database with 1,500 tables, I only audited about a dozen of them carefully (all very large), as they were the ones I felt needed the most attention.

However you decide to audit the indexes in the databases you manage, you need to come up with a sound plan and carry it out in a systematic way.

As you may have already noticed, the audit checklist I have provided above is not long. This is intentional. Remember, the goal of this article series on doing a performance audit is to identify the "easy" and "obvious" performance issues, not to find them all. The ones that I have listed above will get you a long way to identifying and correcting the easy index-related performance problems. Once you have gotten these out of the way, then you can spend time on tougher ones. For example, this website has many index-related tips, many of them very advanced, on these topics:

- [Indexes \(General\)](#)
- [Indexes \(Clustered\)](#)
- [Indexes \(Composite\)](#)
- [Indexes \(Covering\)](#)
- [Indexes \(Non-clustered\)](#)

- [Indexes \(Rebuilding\)](#)
- [Index Tuning Wizard](#)

If you have not done so yet, you will want to review each of these tips pages.

### **Have You Run the Index Tuning Wizard Recently?**

One of the best tools that Microsoft has given us in SQL Server 7.0 and 2000 is the Index Tuning Wizard. It is not a perfect tool, but it does help you to identify existing indexes that aren't being used, along with recommending new indexes that can be used to help speed up queries. If you are using SQL Server 2000, it can also recommend the use of Indexed Views. It uses the actual queries you are running in your database, so its recommendations are based on how your database is really being used. The queries it needs for analysis come from the SQL Server Profiler traces you create.

One of the first things I do when doing a performance audit on a new SQL Server is to capture a trace of server activity and run the Index Tuning Wizard against it. In many cases, it can help me to quickly identify any indexes that are not being used and can be deleted, and to identify new indexes that should be added in order to boost the database's performance.

Here are some tips for using the Index Tuning Wizard when auditing a SQL Server database's indexes:

- When you do the Profiler capture (which is used by the Index Tuning Wizard to perform its analysis), capture the data during a time of day that is representative of a normal load on the database. I generally like to pick a time during mid-morning or mid-afternoon, and then run the Profiler trace over a period of one hour.
- Once the Profiler trace has been captured, the Index Tuning Wizard can be run at any time. But, it is a good idea to run it when the database is not busy, preferably after hours. This is because the analysis performed by the Index Tuning Wizard incurs some server overhead, and there is no point in negatively affecting the server's performance if you don't have to. Also, avoid running the analysis on your production server (the Wizard will still have to connect to the production server), but running the Wizard on another server reduces the load on the production server when the analysis is performed.
- Although it will take more time for the analysis to complete, you need to specify during the setup of the Index Tuning Wizard several options that will help ensure a thorough analysis. These include: not selecting the option to "Keep all existing indexes," as you will want to identify those indexes that are not being used; specifying that you want to perform a "Thorough" analysis, not a "Fast" or "Medium" one; not selecting the option to "Limit the number of workload queries to sample," and to leave the "maximize columns per index" setting to its maximum setting of 16; and specifying that all tables are to be selected for tuning. By selecting these options, you allow the Index Tuning Wizard to do its job thoroughly, although it might take hours for it to complete, depending on the size of the Profiler trace and the speed of hardware you are performing the analysis on. Note: these instructions are for SQL Server 2000, SQL Server 7.0 instructions are slightly different.
- Once the analysis is complete, the Wizard might not have any recommendations, it may recommend to remove one or more indexes, or it may recommend to add one or more indexes, or it may recommend both. You will need to carefully evaluate its recommendations before you take them. For example, the Wizard might recommend to drop a particular index, but you know that this particular index is really needed. So why did the Wizard recommend it be deleted when you know it is not a good idea? This is because the Wizard does not analyze every query found in the trace file (only a sample of them), plus it is possible that your sample trace data did not include the query that needs the index. In these cases, the Wizard might recommend that an index be dropped, even though it may not be a good idea. Once you verify that an index is not needed, should you drop it.

If the Wizard recommends adding new indexes, you will want to evaluate them, and also compare them to the currently existing indexes on the table to see if they make sense and might potentially cause new problems. For example, a recommended index might help a particular query, but it may also slow down a common INSERT operation this is performed thousands of times each hour. The Wizard can't know this, and you must decide what is more important, some queries that run a little faster and INSERTs that run a little slower, or vice versa.

And last of all, even if the Index Tuning Wizard doesn't recommend any new indexes, this doesn't mean that no new indexes are needed, only that based on the trace data that was analyzed that it didn't recommend any. You might want to consider running several traces over several days in order to get an even wider sample of the data in order to better help identify necessary indexes. And even then, the Index Tuning Wizard can't find all the needed indexes, but it will find all the obviously needed ones.

Once you have performed your analysis and made the recommended changes, I recommend that you do another trace and analysis in order to see what affect your changes made. Also keep in mind that using the Index Wizard Analysis is not a one time event. The underlying data in a database changes over time, along with the types of queries run. So you should make it a point to take traces and run analyses periodically on your servers to keep them in regular tune.

### **Does Every Table in Each Databases Have a Clustered Index?**

As a rule of thumb, every table in every database should have a clustered index. Generally, but not always, the clustered index should be on a column that monotonically increases--such as an identity column, or some other column where the value is increasing--and is unique. In many cases, the primary key is the ideal column for a clustered index.

If you have any experience with performance tuning SQL Server 6.5, you may have heard that is not a good idea to add a clustered index to a column that monotonically increases because it can cause a "hotspot" on the disk that can cause performance problems. That advice is true in SQL Server 6.5.

In SQL Server 7.0 and 2000, "hotspots" aren't generally a problem. You would have to have over 1,000 transactions a second before a "hotspot" were to negatively affect performance. In fact, a "hotspot" can be beneficial under these circumstances because it eliminates page splits.

Here's why. If you are inserting new rows into a table that has a clustered index as its primary key, and the key monotonically increases, these means that each INSERT will physically occur one after another on the disk. Because of this, page splits won't occur, which in itself saves overhead. This is because SQL Server has the ability to determine if data being inserted into a table has a monotonically increasing sequence, and won't perform page splits when this happens.

If you are inserting a lot of rows into a heap (a table without a clustered index), data is not inserted in any particular order onto data pages, whether the data is monotonically or not monotonically increasing. This results in SQL Server having to work harder (more reads) to access the data when requested from disk. On the other hand, if a clustered index is added to a table, data is inserted sequentially on data pages, and generally less disk I/O is required to retrieve the data when requested from disk.

If data is inserted into a clustered index in more or less random order, data is often inserted randomly into data pages, which is similar to the problem of inserting data into a heap, which contributes to page splits.

So again, the overall best recommendation is to add a clustered index to a column that monotonically increases (assuming there is a column that does so), for best overall performance. This is especially true if the table is subject to many INSERTS, UPDATES, and DELETES. But if a table is subject to few data modification, but to many SELECT statements, then this advice is less useful, and other options for the clustered index should be considered.

As part of your index audit, check to see if every table in your databases has an index or not. If there is no index at all, seriously consider adding a clustered index, based on the advice provided above. There is virtually no downside to adding a clustered index to a table that does not already have one.

### **Are Any of the Columns in Any Table Indexed More than Once?**

This may sound like obvious advice, but it is more common than you think, especially if a database has been around for awhile and it has been administered by multiple DBAs. SQL Server doesn't care if you do this, as long as the names of the indexes are different. So as you examine your table's current indexes, check to see if any columns have unnecessary duplicate indexes. Removing them not only reduces disk space, but speeds up all data access or modification to that table.

One common example of duplicate indexes is forgetting that columns that have a primary key, or that are specified as unique, are automatically indexed, and then indexing them again under different index names.

### **Are There Any Indexes that are Not Being Used in Queries?**

This is another obvious piece of advice, but is also a common problem, especially if the initial indexes created for the database were "guessed at" by DBAs or developers before the database went into production. Just looking at a table's indexes won't tell you if they index is being used or not, so identifying unused indexes is not always easy.

One of the best ways to identify unused indexes is to use the Index Tuning Wizard, which was previously discussed.

Unnecessary indexes, just like duplicate indexes, wastes disk space and contribute to less than optimal data access and modification performance.

## Are the Indexes too Wide?

The wider an index, the bigger the index becomes physically, and the more work SQL Server has to perform when accessing or modifying data. Because of this, you should avoid adding indexes to very wide columns. The narrower the index, the faster it will perform.

In addition, composite indexes, that include two or more columns, also present the same potential problem. Generally, composite indexes should be avoided, if at all possible. Often, the heavy use of composite indexes in a database means that the database design is flawed.

You can't always avoid indexing wide columns or using composite indexes, but if you think you have to use one, be sure you have carefully evaluated your choice and are confident you don't have other options that may offer better performance.

## Are Tables That are JOINed Have the Appropriate Indexes on the JOINed Columns?

In essence, the column (or columns) used in tables being JOINed should be indexed for best performance. This is straight-forward advice and fairly obvious, but auditing your indexes for optimal JOIN performance is not easy, as you must be familiar with all the JOINS being performed in your database in order to fully perform the audit.

Many people, when creating primary key/foreign key relationships (which are often used in JOINS) forget that while an index is automatically created on the primary key column(s) when it is established, an index for a foreign key is not automatically created, and must be created manually if there is to be one.

Because this is often forgotten, as part of your audit, you may want to identify all primary key/foreign key relationships in your tables and then verify that each foreign key column has an appropriate index.

Besides this, you can also use the Index Tuning Wizard can help identifying missing JOIN indexes, but I have found that the Wizard doesn't always identify missing indexes for JOINed tables. When it comes right down to it, unless you know the types of common JOINS being run against your database, it is tough to identify all the columns that could benefit from an appropriate index.

## Are the Indexes Unique Enough to be Useful?

Just because a table has one or more indexes doesn't mean that the SQL Server Query Analyzer will use them. Before they are used, the Query Optimizer has to consider them useful. If a column in a table is not at least 95% unique, then most likely the query optimizer will not use an available non-clustered index based on that column. Because of this, don't add non-clustered indexes to columns that aren't at least 95% unique. For example, a column with "yes" or "no" as the data won't be at least 95% unique, and creating an index on that column would in essence create an index that would never be used, which we have already learned puts a drag on performance.

As part of your audit, consider "eye-balling" the data in your tables. In other words, take a look at the data residing in your tables, and take an extra look at the columns that are indexed. Generally, it is very obvious if the data in a column is selective or not. If you notice that the data is all "male or female," "y" or "n," and so on, then this data is not selective and any indexes created on them will be a waste of space and a hindrance to performance.

## Are You Taking Advantage of Covering Indexes?

A covering index, which is a form of a composite index, includes all of the columns referenced in the SELECT, JOIN, and WHERE clauses of a query. Because of this, the index contains the data you are looking for and SQL Server doesn't have to look up the actual data in the table, reducing logical and/or physical I/O, thus boosting performance. While non-covering composite indexes can hinder performance, covering composite indexes can be very useful, and in many cases, really boost the performance of a query.

The hard part is to identify where covering indexes might be best used. While the Index Tuning Wizard will help, it will still miss a lot of opportunities where covering indexes can be useful. Otherwise, the only way to identify if they will be useful is to carefully examine all the common queries that run across your database, which of course is near impossible, unless you are really, really bored and have nothing better to do.

At this point, in your audit, the goal should not be to identify new covering indexes as such, but to be aware of them so you can take advantage of them when you run across a situation where they will be helpful.

## How Often are Indexes Rebuilt?

Over time, indexes get fragmented, which causes SQL Server to work harder in order to access them, hurting performance. The only solution to this is to defragment all indexes in your databases on a regular basis. There are a variety of ways to do this, and the how-to process won't be discussed here, as this is explained elsewhere on this website and in the SQL Server Books Online.

The goal of your audit is to find out whether or not the indexes in the databases you are auditing are being defragmented on a regular basis. How often you defragment them can range from daily, weekly, or even monthly, and depends on how often modifications are done, along with the size of the database. If a database has many modifications made daily, then defragmentation should be performed more often. If a database is very large, this means the defragmentation will take longer, which may mean that it cannot be performed as often because the defragmentation process takes too many resources and negatively affects users. As part of your audit, you may want to also evaluate how often the fragmentation is currently being done, and to find out if this is the optimal frequency.

At the very least, if indexes aren't currently being rebuilt now, they need to be, and as part of your audit, you need to ensure that some sort of an appropriate index rebuilding scheme is put into place.

### **What is Your Index Fillfactor?**

Closely related to index rebuilding is the fillfactor. When you create a new index, or rebuild an existing index, you can specify a fillfactor, which is the amount the data pages in the index are filled when they are created. A fillfactor of 100 means that each index page is 100% full, a fillfactor of 50% means each index page is 50% full.

If you create a clustered index (on a non-monotonically ascending column) that has a fillfactor of 100, that means that each time a record is inserted (or perhaps updated), page splits will occur because there is no room for the data in the existing pages. Numerous page splits can slow down SQL Server's performance.

Here's an example: Assume that you have just created a new index on a table with the default fillfactor. When SQL Server creates the index, it places the index on contiguous physical pages, which allows optimal I/O access because the data can be read sequentially. But as the table grows and changes with INSERTS, UPDATES, and DELETES, page splitting occurs. When pages split, SQL Server must allocate new pages elsewhere on the disk, and these new pages are not contiguous with the original physical pages. Because of this, random I/O, not sequential I/O access must be used, which is much slower, to access the index pages.

So what is the ideal fillfactor? It depends on the ratio of reads to writes that your application makes to your SQL Server tables. As a rule of thumb, follow these guidelines:

- Low Update Tables (100-1 read to write ratio): 100% fillfactor
- High Update Tables (where writes exceed reads): 50%-70% fillfactor
- Everything In-Between: 80%-90% fillfactor.

You may have to experiment to find the optimum fillfactor for your particular application. Don't assume that a low fillfactor is always better than a high fillfactor. While page splits will be reduced with a low fillfactor, it also increases the number of pages that have to be read by SQL Server during queries, which reduces performance. And not only is I/O overhead increased with a too low of fillfactor, it also affects your buffer cache. As data pages are moved in from disk to the buffer, the entire page (including empty space) is moved to the buffer. So the lower the fillfactor, the more pages that have to be moved into SQL Server's buffer, which means there is less room for other important data pages to reside at the same time, which can reduce performance.

If you don't specify a fillfactor, the default fillfactor is 0, which means the same as a 100% fillfactor, (the leaf pages of the index are filled 100%, but there is some room left on intermediate index pages).

As part of your audit process, you need to determine what fillfactor is being used to create new indexes and rebuild current indexes. In virtually all cases, except for read-only databases, the default value of 0 is not appropriate. Instead, you will want a fillfactor that leaves an appropriate amount of free space, as discussed above.

## Tips on Optimizing SQL Server Indexes

**All the tips provided on this website about indexing are general guidelines.** As with any general guideline, there are exceptions. Because of this, it is a good idea to test out various indexing strategies for the most common queries run against your database. Only by testing different strategies against your queries can you be sure that you have fully optimized your database.

\*\*\*\*\*

If your databases are set for "Auto Create Statistics," the SQL Server Query Optimizer, when running queries, will consider if there is any benefit to adding statistics for any column that doesn't already have statistics for it. This is for columns without indexes. This is a good thing as it helps provide better information to the Query Optimizer so that better optimized execution plans are created to execute queries.

**The addition of an automatically added statistics to a column is also useful for something else. It is a clue to the potential need for an index on the column.** In other words, if the Query Optimizer thinks that column statistics are useful, there is also a good chance that adding an appropriate index to this same column would be useful. This is not always the case, so you will need to perform some testing--before and after an index is added--to see if adding an index actually helps or not. But this is simple to do.

How do you know if the Query Optimizer has automatically created column statistics on a column in a table? Actually, this is quite easy to find out. Run the following query from Query Analyzer, which is pointing to a user database.

```
SELECT name
FROM sysindexes
WHERE (name LIKE '%_WA_Sys%')
```

This query will return all of the columns from the tables in your database that have column statistics on them that have been added automatically by the Query Optimizer. The value that is in the "name" column of the sysindexes table is the name assigned to the statistics that SQL Server keeps track of for the named column. This information provide you a starting point from which to explore whether or not adding indexes to these columns will be useful or not.

\*\*\*\*\*

**An index on a column can often be created different ways, some of which are more optimal than others.** What this means that just because you create a useful index on a column doesn't mean that it automatically is the optimum version of that index. It is quite possible that a different variation of the same index is faster.

The most obvious example of this is that an index can be a clustered or non-clustered. Another example of how an index is created that can affect its performance is the FILLFACTOR and PAD\_INDEX settings used to create it. Also, whether the index is also a composite index or not (and what columns it contains) can affect an index's performance.

Unfortunately, there is no easy answer as to which variation of the same index is the fastest in your situation, as the data and queries run against the data are different.

While I can't offer you specific rules that fit in all cases, the index tips you find on this website should help you decide which variation of an index is best in your particular circumstance. You may also need to test variations of the same index to see which variation works best for you.

\*\*\*\*\*

Indexes cannot be created in a vacuum. In other words, **before you can identify and create optimal indexes for your tables, you must thoroughly understand the kinds of queries that will be run against them.** This is not an easy task, especially if your are attempting to add indexes to a new database.

Whether you are optimizing the indexes for the first time for a new database, or for a current production database, you need to identify what queries are run, and how often they are run. Obviously, you will want to spend more time creating and tuning indexes for queries that are run very often than for queries that are seldomly run. In addition, you will want to identify those queries that are the most resource intensive, even if they aren't run the most often.

Once you know which queries run the most often, and which are the most resource intensive, you can begin to better allocate your time in order to get the biggest bang for your time invested.

But there is still one little question. How do you identify which queries are run the most often, and which are the most resource intensive? The easiest solution is to capture a Profiler trace, which can be configured to identify which queries run the most often, and to identify which queries use the most resources. How you configure Profiler won't be discussed now, as it would take a large article to explain all the options. The point here is to make you aware that the Profiler is the tool of choice to identify the queries that are being run against your database.

If you are adding indexes to a production database, capturing the data you need is simple. But if your database is new, what you will need to do is to simulate actual activity as best as possible, perhaps during beta testing of the application, and capture this activity. While it may not be perfect data, it will at least give you a head start. And once production begins, you can continue your index tuning efforts on an on-going basis until you are relatively satisfied that you have identified and tuned the indexes to the best of your ability.

Once you have identified the key queries, your next job is to identify the best indexes for them. This is also a process too big to describe in this single tip, although there are many tips on this website that relate directly to this issue. Essentially, what you need to do is to run each query you need to analyze in Query Analyzer, examining how it works, and examining its execution plan. Then based on your knowledge of the query, and your knowledge of indexing and how it works in SQL Server, you begin the art of adding and optimizing indexes for your application.

\*\*\*\*\*

**Indexes should be considered on all columns that are frequently accessed by the WHERE, ORDER BY, GROUP BY, TOP, and DISTINCT clauses.** Without an index, each of these operations will require a table scan of your table, potentially hurting performance.

Keep in mind the word "considered". An index created to support the speed of a particular query may not be the best index for another query on the same table. Sometimes you have to balance indexes to attain acceptable performance on all the various queries that are run against a table.

\*\*\*\*\*

**Don't automatically add indexes** on a table because it seems like the right thing to do. Only add indexes if you know that they will be used by the queries run against the table.

\*\*\*\*\*

**As a rule of thumb, every table should have at least a clustered index.** Generally, but not always, the clustered index should be on a column that monotonically increases--such as an identity column, or some other column where the value is increasing--and is unique. In many cases, the primary key is the ideal column for a clustered index. See this [url](#) for more details about clustered indexes.

\*\*\*\*\*

**Static tables** (those tables that change very little, or not at all) can be more heavily indexed than dynamic tables (those that are subject to many INSERTs, UPDATES, or DELETES) without negative effect. This doesn't mean you should index every column. Only those columns that need an index should have them. But at least you don't have to worry about the overhead of indexes when they are added to static tables, as you must keep in mind when adding indexes to dynamic tables.

In addition for these tables, create the indexes with a FILLFACTOR and a PAD\_INDEX of 100 to ensure there is no wasted space. This reduces disk I/O, helping to boost overall performance.

\*\*\*\*\*

**Point queries, queries that return a single row, are just as fast using a clustered index as a non-clustered index.** If you will be creating an index to speed the retrieval of a single record, you may want to consider making it a non-clustered index, and saving the clustering index (you can only have one) for a more complex query.

\*\*\*\*\*

To help **identify which tables in your database may need additional or improved indexes**, use the SQL Server Profiler Create Trace Wizard to run the "Identify Scans of Large Tables" trace. This trace will tell which tables are being scanned by queries instead of using an index to seek the data. This should provide you data you can use to help you identify which tables may need additional or better indexes.

\*\*\*\*\*

**Don't over index your OLTP tables**, as every index you add increases the time it takes to perform INSERTS, UPDATES, and DELETES. There must be a fine line drawn between having the ideal number of indexes (for SELECTs) and the ideal number for data modifications.

\*\*\*\*\*

**Don't accidentally add the same index twice on a table.** This is easier to happen than you think. For example, you add a unique or primary key to a column, which of course creates an index to enforce what you want to happen. But without thinking about it when evaluating the need for indexes on a table, you decide to add a new index, and this new index happens to be on the same column as the unique or primary key. As long as you give indexes different names, SQL Server will allow you to create the same index over and over.

\*\*\*\*\*

**Drop indexes that are never used by the Query Optimizer.** Unused indexes slow data modifications, causes unnecessary I/O reads when reading pages, and wastes space in your database, increasing the amount of time it takes to backup and restore databases. Use the Index Wizard (7.0 and 2000) to help identify indexes that are not being used.

\*\*\*\*\*

Generally, **you probably won't want to add an index to a table under these conditions:**

- If the index is not used by the query optimizer. Use Query Analyzer's "Show Execution Plan" option to see if your queries against a particular table use an index or not. If the table is small, most likely indexes will not be used.
- If the column values exhibit low selectivity, often less than 90%-95% for non-clustered indexes.
- If the column(s) to be indexed are very wide.
- If the column(s) are defined as TEXT, NTEXT, or IMAGE data types.
- If the table is rarely queried.

Creating an index under any of these conditions will most likely results in an index that is rarely, no not used at all.

\*\*\*\*\*

While high index selectivity is generally an important factor that the Query Optimizer uses to determine whether or not to use an index, there is one special case where indexes with low selectivity can be useful speeding up SQL Server. This is the case for indexes on foreign keys.

**Whether an index on a foreign key has either high or low selectivity, an index on a foreign key can be used by the Query Optimizer to perform a merge join on the tables in question.** A merge join occurs when a row from each table is taken and then they are compared to see if they match the specified join criteria. If the tables being joined have the appropriate indexes (no matter the selectivity), a merge join can be performed, which is generally much faster than a join to a table with a foreign key that does not have an index.

\*\*\*\*\*

On **data warehousing databases**, which are essentially read-only, having an many indexes as necessary for covering virtually any query is not normally a problem.

\*\*\*\*\*

**To provide the up-to-date statistics the query optimizer needs to make smart query optimization decisions, you will generally want to leave the "Auto Update Statistics" database option on.** This helps to ensure that the optimizer statistics are valid, helping to ensure that queries are properly optimized when they are run.

But this option is not a panacea. When a SQL Server database is under very heavy load, sometimes the auto update statistics feature can update the statistics at inappropriate times, such as the busiest time of the day.

If you find that the auto update statistics feature is running at inappropriate times, you may want to turn it off, and then manually update the statistics (using UPDATE STATISTICS or sp\_updatestats) when the database is under a less heavy load.

But again, consider what will happen if you do turn off the auto update statistics feature? While turning this feature off may reduce some stress on your server by not running at inappropriate times of the day, it could also cause some of your queries not to be properly optimized, which could also put extra stress on your server during busy times.

Like many optimization issues, you will probably need to experiment to see if turning this option on or off is more effective for your environment. But as a rule of thumb, if your server is not maxed out, then leaving this option on is probably the best decision.

\*\*\*\*\*

Keep the **"width" of your indexes as narrow as possible**. This reduces the size of the index and reduces the number of disk I/O reads required to read the index, boosting performance.

\*\*\*\*\*

If possible, try to create indexes on columns that have **integer values** instead of characters. Integer values have less overhead than character values.

\*\*\*\*\*

**If you have two or more tables that are frequently joined together**, then the columns used for the joins should have an appropriate index. If the columns used for the joins are not naturally compact, then considering adding surrogate keys to the tables that are compact in order to reduce the size of the keys, thus decreasing I/O during the join process, which increases overall performance.

\*\*\*\*\*

**When creating indexes, try to make them unique indexes** if at all possible. SQL Server can often search through a unique index faster than a non-unique index because in a unique index, each row is unique, and once the needed record is found, SQL Server doesn't have to look any further.

\*\*\*\*\*

If a particular **query against a table is run infrequently**, and the addition of an index greatly speeds the performance of the query, but the performance of INSERTS, UPDATES, and DELETES is negatively affected by the addition of the index, consider creating the index for the table for the duration of when the query is run, then dropping the index. An example of this is when monthly reports are run at the end of the month on an OLTP application.

\*\*\*\*\*

If you like to get under the cover of SQL Server to learn more about indexing, take a look at the **sysindex system table** that is found in every database. Here, you can find a wealth of information on the indexes and tables in your database. To view the data in this table, run this query from the database you are interested in: SELECT \* FROM sysindexes. Here are some of the more interesting fields found in this table:

- *dpages*: If the indid value is 0 or 1, then dpages is the count of the data pages used for the index. If the indid is 255, then dpages equals zero. In all other cases, dpages is the count of the non-clustered index pages used in the index.
- *id*: Refers to the id of the table this index belongs to.
- *indid*: This column indicates the type of index. For example, 1 is for a clustered table, a value greater than 1 is for a non-clustered index, and a 255 indicates that the table has text or image data.
- *OrigFillFactor*: This is the original fillfactor used when the index was first created, but it is not maintained over time.
- *statversion*: Tracks the number of times that statistics have been updated.

- *status*: 2 = unique index, 16 = clustered index, 64 = index allows duplicate rows, 2048 = the index is used to enforce the Primary Key constraint, 4096 = the index is used to enforce the Unique constraint. These values are additive, and the value you see in this column may be a sum of two or more of these options.
- *used*: If the *indid* value is 0 or 1, then *used* is the number of total pages used for all index and table data. If *indid* is 255, *used* is the number of pages for text or image data. In all other cases, *used* is the number of pages in the index.

\*\*\*\*\*

**Avoid using FLOAT or REAL data types as primary keys**, as they add unnecessary overhead that can hurt performance.

\*\*\*\*\*

If you want to **boost the performance of a query that includes an AND operator** in the WHERE clause, consider the following:

- Of the search criterions in the WHERE clause, at least one of them should be based on a highly selective column that has an index.
- If at least one of the search criterions in the WHERE clause is not highly selective, consider adding indexes to all of the columns referenced in the WHERE clause.
- If none of the column in the WHERE clause are selective enough to use an index on their own, consider creating a covering index for this query.

\*\*\*\*\*

The **Query Optimizer will always perform a table scan or a clustered index scan** on a table if the WHERE clause in the query contains an OR operator and if any of the referenced columns in the OR clause are not indexed (or does not have a useful index). Because of this, if you use many queries with OR clauses, you will want to ensure that each referenced column in the WHERE clause has an index.

\*\*\*\*\*

**A query with one or more OR clauses can sometimes be rewritten as a series of queries that are combined with a UNION statement** in order to boost the performance of the query. For example, let's take a look at the following query:

```
SELECT employeeID, firstname, lastname
FROM names
WHERE dept = 'prod' or city = 'Orlando' or division = 'food'
```

This query has three separate conditions in the WHERE clause. In order for this query to use an index, then there must be an index on all three columns found in the WHERE clause.

This same query can be written using UNION instead of OR, like this example:

```
SELECT employeeID, firstname, lastname FROM names WHERE dept = 'prod'
UNION ALL
SELECT employeeID, firstname, lastname FROM names WHERE city = 'Orlando'
UNION ALL
SELECT employeeID, firstname, lastname FROM names WHERE division = 'food'
```

Each of these queries will produce the same results. If there is only an index on *dept*, but not the other columns in the WHERE clause, then the first version will not use any index and a table scan must be performed. But in the second version of the query will use the index for part of the query, but not for all of the query.

Admittedly, this is a very simple example, but even so, it does demonstrate how rewriting a query can affect whether or not an index is used or not. If this query was much more complex, then the approach of using UNION might be must more efficient, as it allows you to tune each part of the index individually, something that cannot be done if you use only ORs in

your query.

If you have a query that uses ORs and it not making the best use of indexes, consider rewriting it as a UNION, and then testing performance. Only through testing can you be sure that one version of your query will be faster than another.

\*\*\*\*\*

**The Query Optimizer converts the Transact-SQL IN clause to the OR operator** when parsing your code. Because of this, keep in mind that if the referenced column in your query doesn't include an index, then the Query Optimizer will perform a table scan or clustered index scan on the table.

\*\*\*\*\*

**If you use the SOUNDEX function against a table column in a WHERE clause, the Query Optimizer will ignore any available indexes and perform a table scan.** If your table is large, this can present a major performance problem. If you need to perform SOUNDEX type searches, one way around this problem is to pre-calculate the SOUNDEX code for the column you are searching and then place this value in a column of its own, and then place an index on this column in order to speed searches.

\*\*\*\*\*

**If you need to create indexes on large tables in SQL Server 2000,** you may be able to speed up their creation by using the SORT\_IN\_TEMPDB option available with the CREATE INDEX command. This option tells SQL Server to use the tempdb database, instead of the current database, to sort data while creating indexes.

Assuming your tempdb database is isolated on its own separate disk or disk array, then the process of creating the index can be sped up.

The only slight downside to using this option is that it takes up slightly more disk space than if you didn't use it, but this shouldn't be much of an issue in most cases. If your tempdb database is not on its own disk or disk array, then don't use this option, as it can actually slow performance.

\*\*\*\*\*

SQL Server 2000 Enterprise Edition (not the standard edition) offers the **ability to create indexes in parallel**, greatly speeding index creation. Assuming your server has multiple CPUs, SQL Server 2000 uses near-linear scaling to boost index creation speed. For example, using two CPUs instead of one CPU almost halves the speed it takes to create indexes.

\*\*\*\*\*

**SQL Server 2000 offers a function called CHECKSUM.** The main purpose for this function is to create what are called hash indices. A hash indices is an index built on a column that stores the checksum of the data found in another column in the table. The CHECKSUM function takes data from another column and creates a checksum value. In other words, the CHECKSUM function is used to create a mostly unique value that represents other data in your table. In most cases, the CHECKSUM value will be much smaller than the actual value. For the most part, checksum values are unique, but this is not guaranteed. It is possible that two slightly different values may produce the same identical CHECKSUM value.

Here's how this works using our music database example. Say we have a song with the title "My Best Friend is a Mule from Missouri". As you can see, this is a rather long value, and adding an index to the song title column would make for a very wide index. But in this same table, we can add a CHECKSUM column that takes the title of the song and creates a checksum based on it. In this case, the checksum would be 1866876339. The CHECKSUM function always works the same, so if you perform the CHECKSUM function on the same value many different times, you would always get the same result.

So how does the CHECKSUM help us? The advantage of the CHECKSUM function is that instead of creating a wide index by using the song title column, we create an index on the CHECKSUM column instead. "That's fine and dandy, but I thought you wanted to search by the song's title? How can anybody ever hope to remember a checksum value in order to perform a search?"

Here's how. Take a moment to review this code:

```
SELECT title, artist, composer
FROM songs
WHERE title = 'My Best Friend is a Mule from Missouri'
```

```
AND checksum_title = CHECKSUM('My Best Friend is a Mule from Missouri')
```

In this example, it appears that we are asking the same question twice, and in a sense, we are. The reason we have to do this is because there may be checksum values that are identical, even though the names of the songs are different. Remember, unique checksum values are not guaranteed.

Here's how the query works. When the Query Optimizer examines the WHERE clause, it determines that there is an index on the checksum\_title column. And because the checksum\_title column is highly selective (minimal duplicate values) the Query Optimizer decides to use the index. In addition, the Query Optimizer is able to perform the CHECKSUM function, converting the song's title into a checksum value and using it to locate the matching records in the index. Because an index is used, SQL Server can quickly locate the rows that match the second part of the WHERE clause. Once the rows have been narrowed down by the index, then all that has to be done is to compare these matching rows to the first part of the WHERE clause, which will take very little time.

This may seem a lot of work to shorten the width of an index, but in many cases, this extra work will pay off in better performance in the long run.

Because of the nature of this tip, I suggest you experiment using this method, and the more conventional method of creating an index on the title column itself. Since there are so many variables to consider, it is tough to know which method is better in your particular situation unless you give them both a try.

\*\*\*\*\*

Some queries can be very complex, involving many tables, joins, and other conditions. I have seen some queries run over 1000 lines of code (I didn't write them). This can make them difficult to analyze in order to identify what indexes might be used to help the query perform better.

For example, perhaps you want to create a covering index for the query and you need to identify the columns to include in the covering index. Or, perhaps you want to identify those columns that are used in joins in order to check to see that you have indexes on those columns used in the joins in order to maximize performance.

**To make complex queries easier to analyze, consider breaking them down into their smaller constituent parts.** One way to do this is to simply create lists of the key components of the query, such as:

- List all of the columns that are to be returned
- List all of the columns that are used in the WHERE clause
- List all of the columns used in the JOINS (if applicable)
- List all the tables used in JOINS (if applicable)

Once you have the above information organized in this easy-to-comprehend form, it is must easier to identify those columns that could potentially make use of indexes when executed.

\*\*\*\*\*

**Queries that include either the DISTINCT or the GROUP BY clauses can be optimized by including appropriate indexes.** Any of the following indexing strategies can be used:

- Include a covering, non-clustered index (covering the appropriate columns) of the DISTINCT or GROUP BY clauses.
- Include a clustered index on the columns in the GROUP BY clause.
- Include a clustered index on the columns found in the SELECT clause.

Adding appropriate indexes to queries that include DISTINCT or GROUP BY is most important for those queries that run often. If a query is rarely ran, then adding an index for its benefit may cause more performance problems than it prevents.

\*\*\*\*\*

**Computed columns in SQL Server 2000 can be indexed** if they meet *all* of the following criteria:

- The computed column's expression is deterministic. This means that the computed value must always be the same given the same inputs.
- The ANSI\_NULL connection-level object was on when the table was created.
- TEXT, NTEXT, or IMAGE data types are not used in the computed column.
- The physical connection used to create the index, and all connections used to INSERT, UPDATE, or DELETE rows in the table must have these six SET options properly configured: ANSI\_NULLS = ON, ANSI\_PADDINGS = ON, ANSI\_WARNINGS = ON, ARITHABORT = ON, CONCAT\_NULL\_YIELDS\_NULL = ON, QUOTED\_IDENTIFIER = ON, NUMERIC\_ROUNDABORT = OFF.

If you create a clustered index on a computed column, the computed values are stored in the table, just like with any clustered index. If you create a non-clustered index, the computed value is stored in the index, not in the actual table.

While adding an index to a computed column is possible, it is rarely advisable. The biggest problem with doing so is that if the computed column changes, then the index (clustered or non-clustered) has to also be updated, which contributes to overhead. If there are many computed values changing, this overhead can significantly hurt performance.

The most common reason you might consider adding an index to a computed column is if you are using the CHECKSUM() function on a large character column in order to reduce the size of an index. By using the CHECKSUM() of a large character column, and indexing it instead of the large character column itself, the size of the index can be reduced, helping to save space and boost overall performance.

\*\*\*\*\*

**Many databases experience both OLTP and OLAP queries.** As you probably already know, it is nearly impossible to optimize the indexing of a database that has both type of queries. This is because in order for OLTP queries to be fast, there should not be too many indexes as to hinder INSERT, UPDATE, or DELETE operations. And for OLAP queries to be fast, there should be as many indexes as needed to speed SELECT queries.

While there are many options for dealing with this dilemma, one option that may work for some people is a strategy where OLAP queries are mostly (if not all) are run during off hours (assuming the database has any off hours), and take advantage of indexes that are added each night before the OLAP queries begin, and then are dropped once the DSS queries are complete. This way, those indexes needed for fast performing OLAP queries will minimally interfere with OLTP transactions (especially during busy times).

As you can imagine, this strategy can take a lot of planning and work, but in some cases, it can offer the best performance for databases that experience both OLTP and OLAP queries. Because it is hard to guess if this strategy will work for you, you will want to test it before putting it into production.

\*\*\*\*\*

**Be aware that the MIN() or MAX() functions can take advantage of appropriate indexes.** If you find that you are using these functions often, and your current query is not taking advantage of current indexes to speed up these functions, consider adding appropriate indexes.

\*\*\*\*\*

**If you know that a particular column will be subject to many sorts,** consider adding a unique index to that column. This is because unique columns generally sort faster in SQL Server than if there are duplicate column data present.

\*\*\*\*\*

**Whenever you upgrade software that affects SQL Server,** be sure that you rerun the Index Wizard to catch any obvious missing indexes. Application software that is updated often changes the Transact-SQL code (and SPs, if used) that accesses SQL Server. In many cases, the vendor supplying the upgraded code may not have taken into account how index use might have changed after the upgrade was made. Because of this, it is very important to get a good Profiler trace of the new code interacting with your database, and then use Index Wizard to help identify any missing indexes. You may be surprised what you find.

\*\*\*\*\*

DELETE operations can sometimes be time- and space-consuming. In some environments you might be able to increase the performance of this operation when using TRUNCATE instead of DELETE. TRUNCATE will almost instantly be executed. However, TRUNCATE will not work when there are Foreign Key references present for that table. A workaround is to DROP the constraints before firing the TRUNCATE. Here's a **generic script that will drop all existing Foreign Key constraints on a specific table**:

```
CREATE TABLE dropping_constraints
(
cmd VARCHAR(8000)
)

INSERT INTO dropping_constraints
SELECT
'ALTER TABLE [' +
t2.Table_Name +
'] DROP CONSTRAINT ' +
t1.Constraint_Name
FROM
INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS t1
INNER JOIN
INFORMATION_SCHEMA.CONSTRAINT_COLUMN_USAGE t2
ON
t1.CONSTRAINT_NAME = t2.CONSTRAINT_NAME
WHERE t2.TABLE_NAME='your_tablename_goes_here'
DECLARE @stmt VARCHAR(8000)
DECLARE @rowcnt INT
SELECT TOP 1 @stmt=cmd FROM dropping_constraints
SET @rowcnt=@@ROWCOUNT
WHILE @rowcnt<>0
BEGIN
EXEC (@stmt)
SET @stmt = 'DELETE FROM dropping_constraints WHERE cmd ='+ QUOTENAME(@stmt, ''')
EXEC (@stmt)
SELECT TOP 1 @stmt=cmd FROM dropping_constraints
SET @rowcnt=@@ROWCOUNT
END
DROP TABLE dropping_constraints
```

This can also be extended to drop all FK constraints in the current database. To achieve this, just comment the WHERE clause.

## Tips on Optimizing SQL Server Clustered Indexes

**As a rule of thumb, every table should have a clustered index.** Generally, but not always, the clustered index should be on a column that monotonically increases--such as an identity column, or some other column where the value is increasing--and is unique. In many cases, the primary key is the ideal column for a clustered index.

If you have any experience with performance tuning SQL Server 6.5, you may have heard that is not a good idea to add a clustered index to a column that monotonically increases because it can cause a "hotspot" on the disk that can cause performance problems. That advice is true in SQL Server 6.5.

In SQL Server 7.0 and 2000, "hotspots" aren't generally a problem. You would have to have over 1,000 transactions a second before a "hotspot" were to negatively affect performance. In fact, a "hotspot" can be beneficial under these circumstances because it eliminates page splits.

Here's why. If you are inserting new rows into a table that has a clustered index as its primary key, and the key monotonically increases, this means that each INSERT will physically occur one after another on the disk. Because of this, page splits won't occur during INSERTs, which in itself saves overhead. This is because SQL Server has the ability to determine if data being inserted into a table has a monotonically increasing sequence, and won't perform page splits when this happens.

If you are inserting a lot of rows into a heap (a table without a clustered index), data is not inserted in any particular order onto data pages, whether the data is monotonically or not monotonically increasing. This results in SQL Server having to work harder (more reads) to access the data when requested from disk. On the other hand, if a clustered index is added to a table, data is inserted sequentially on data pages, and generally less disk I/O is required to retrieve the data when requested from disk.

If data is inserted into a clustered index in more or less random order, data is often inserted randomly into physical data pages, which is similar to the problem of inserting any data into a heap.

So again, often, the best overall recommendation is to add a clustered index to a column that monotonically increases (assuming there is a column that does so). This is especially true if the table is subject to many INSERTS, UPDATES, and DELETES. But if a table is subject to few data modification, but to many SELECT statements, then this advice is less useful, and other options for the clustered index should be considered. Read the other tips on this page to learn of situations where you should place the clustered index on other columns.

\*\*\*\*\*

Here are some **more good reasons to add a clustered index to every table.**

Keep in mind that a clustered index physically orders the data in a table based on a single or composite column. Second, the data in a heap (a table without a clustered index) is not stored in any particular physical order.

Whenever you need to query the column or columns used for the clustered index, SQL Server has the ability to sequentially read the data in a clustered index an extent (8 data pages, or 64K) at a time. This makes it very easy for the disk subsystem to read the data quickly from disk, especially if there is a lot of data to be retrieved.

But if a heap is used, even if you add a non-clustered index on an appropriate column or columns, because the data is not physically ordered (unless you are using a covering index), SQL Server has to read the data from disk randomly using 8K pages. This creates a lot of extra work for the disk subsystem to retrieve the same data, hurting performance.

Another disadvantage of a heap is that when you rebuild indexes to reduce fragmentation, heaps are not defragmented, because they are not indexes. This means that over time, a heap will become more and more fragmented, further hurting performance. Adding a cluster index will insure that the table can be defragmented when indexes are rebuilt.

This are just several of many reasons why a clustered index should be added to virtually all tables.

\*\*\*\*\*

Since you **can only create one clustered index** per table, take extra time to carefully consider how it will be used. Consider the type of queries that will be used against the table, and make an educated guess as to which query (the most common one run against the table, perhaps) is the most critical, and if this query will benefit from having a clustered index.

\*\*\*\*\*

**Clustered indexes are useful for queries that meet these specifications:**

- For queries that SELECT by a large range of values or where you need sorted results. This is because the data is already presorted in the index for you. Examples of this include when you are using BETWEEN, <, >, GROUP BY, ORDER BY, and aggregates such as MAX, MIN, and COUNT in your queries.
- For queries that look up a record with a unique value (such as an employee number) and you need to retrieve most or all of the data in the record. This is because the query is covered by the index. In other words, the data you need is in the index itself, and SQL Server does not have to read any additional pages to retrieve the data you want.
- For queries that access columns with a limited number of distinct values, such as a columns that holds country or state data. But if column data has little distinctiveness, such as columns with a yes or no, or male or female, then won't want to "waste" your clustered index on them.
- For queries that use the JOIN or GROUP BY clauses.
- For queries where you want to return a lot of rows, just not a few. Again, this is because the data is in the index and does not have to be looked up elsewhere.

\*\*\*\*\*

If you run into a circumstance where you need to have a single **wide index** (a composite index of three or more columns) in a table, and the rest of the indexes in this table (assuming there are two or more) will only be one column wide, then consider making the wide index a clustered index and the other indexes non-clustered indexes.

Why? If the wide index is a clustered index, this means that the entire table is the index itself, and a large amount of additional disk space is not required to create the index. But if the wide index is a non-clustered index, this means SQL Server will have to create a "relatively large" index, which will indeed consume a large amount of additional disk space. Whenever the index needs to be used by the query processor, it will be more efficient to access the clustered index than the non-clustered index, and performance will be better.

\*\*\*\*\*

**Avoid clustered indexes on columns that are already "covered"** by non-clustered indexes. A clustered index on columns that is already "covered" is redundant. Use the clustered index for columns that can better make use of it.

\*\*\*\*\*

When selecting a column to base your clustered index on, try to **avoid columns that are frequently updated**. Every time that a column used for a clustered index is modified, all of the non-clustered indexes must also be updated, creating additional overhead.

\*\*\*\*\*

**When selecting a column or columns to include in your clustered index**, select the column (or for the first column in a composite index) that contains the data that will most often be searched in your queries.

\*\*\*\*\*

If a **table has both a clustered index and non-clustered indexes**, then performance will be best optimized if the clustered index is based on a single column that is as narrow as possible. This is because non-clustered indexes use the clustered index to locate data rows and because non-clustered indexes must hold the clustered keys within their B-tree structures. This helps to reduce not only the size of the clustered index, but all non-clustered indexes on the table as well.

\*\*\*\*\*

The **primary key you select for your table should not always be a clustered index**. If you create the primary key and don't specify otherwise, this is the default. Only make the primary key a clustered index if you will be regularly performing range queries on the primary key or need your results sorted by the primary key.

\*\*\*\*\*

**If you drop or change a clustered index**, keep in mind that all of the non-clustered indexes also have to be rebuilt. Also keep in mind that to recreate a new clustered index, you will need free disk space equivalent to 1.2 times the size of the table you are working with. This space is necessary to recreate the entire table while maintaining the old table until the new table is recreated. Then the old table is deleted.

\*\*\*\*\*

When deciding on whether to create a clustered or non-clustered index, it is often helpful to **know what the estimated size of the clustered index** will be. Sometimes, the size of a clustered index on a particular column or columns may be very large, leading to database size bloat.

\*\*\*\*\*

Clustered index values often repeat many times in a table's non-clustered storage structures, and a large clustered index value can unnecessarily increase the physical size of a non-clustered index. This increases disk I/O and reduces performance when non-clustered indexes are accessed.

Because of this, **ideally a clustered index should be based on a single column** (not multiple columns) that is as narrow as possible. This not only reduces the clustered index's physical size, it also reduces the physical size of non-clustered indexes and boosts SQL Servers overall performance.

\*\*\*\*\*

When you create a clustered index, **try to create it as a UNIQUE clustered index**, not a non-unique clustered index. The reason for this is that while SQL Server will allow you to create a non-unique clustered index, under the surface, SQL Server will make it unique for you by adding a 4-byte "uniqueifier" to the index key to guarantee uniqueness. This only serves to increase the size of the key, which increases disk I/O, which reduces performance. If you specify that your clustered index is UNIQUE when it is created, you will prevent this unnecessary overhead.

\*\*\*\*\*

**If possible, avoid adding a clustered index to a GUID column** (uniqueidentifier data type). GUIDs take up 16-bytes of storage, more than an Identify column, which in turn make the index larger, which increases I/O reads, which can hurt performance. While the performance hit will be minimal if you do decide to add a clustered index to a GUID column, every little bit adds up.

## Tips on Optimizing SQL Server Composite Indexes

**A composite index is an index that is made up of more than one column.** In some cases, a composite index is also a covering index. Generally speaking, composite indexes (with the exception of covering indexes) should be avoided. This is because composite indexes tend to be wide, which means that the index will be larger, requiring more disk I/O to read it, hurting performance.

The main reason composite indexes (not covering composite indexes) are used is to make an index more selective. A better way to ensure selectivity is good database design, not creating composite indexes.

\*\*\*\*\*

If you have no choice but to use a composite index, keep the **"width" of it as narrow as possible**. This reduces the size of the index and reduces the number of disk I/O reads required to read the index, boosting performance.

\*\*\*\*\*

**A composite index is generally only useful to a query if the WHERE clause of the query matches the column(s) that are leftmost in the index.** So if you create a composite index, such as "City, State", then a query such as "WHERE City = 'Springfield'" will use the index, but the query "WHERE STATE = 'MO'" will not use the index.

\*\*\*\*\*

Even if the WHERE clause in a query does not specify the first column of an available index (which normally disqualifies the index from being used), if the **index is a composite index and contains all of the columns referenced in the query**, the query optimizer can still use the index, because the index is a covering index.

\*\*\*\*\*

When you create an **index with a composite key**, the order of the columns in the key is important. Try to order the columns in the key to enhance selectivity, with the most selective columns to the leftmost of the key. If you don't do this, and put a non-selective column at the first part of the key, you risk having the Query Optimizer not use the index at all. Generally, a column should be at least 95% unique in order for it to be considered selective. [More info here](#).

\*\*\*\*\*

Sometimes, it is a good idea to **split a composite index into multiple single-column indexes**. This is because only the first column in a composite index has statistics stored for it. And if this first column is not very selective, it may not be used by the Query Optimizer. But if you were to break up the composite index into multiple indexes, then statistics will be kept for each column, and the Query Optimizer will have better information to make better decisions on how to use indexes. SQL Server has the ability to join two or more individual indexes and intersect them, just as if you were using a composite index, giving you the best benefits of single and composite indexes.

This is not to say that multiple single indexes is always better than a single composite index. Only through testing will you know for sure which strategy will offer the best performance in your particular circumstance.

\*\*\*\*\*

As you may know, **an index is automatically created for column(s) in your table that you specify as a PRIMARY KEY or as UNIQUE**. If two or more columns are involved, and a composite index is created, you should choose how the columns will be ordered in the composite index, instead of accepting the default choices offered by SQL Server.

This is because you always want to use the most selective columns at the left of the key to ensure that the composite index is selective enough to be used by the Query Optimizer. If you don't do this, then the default order of the columns in the composite index may not be very selective, and the index may not be used by Query Optimizer.

## Not All SQL Server Indexes Are Created Equal

If you have much experience with indexes at all, you are probably already familiar with the difference between clustered and non-clustered indexes. But this article is not about them. This article is about whether or not the SQL Server Query Optimizer will use your carefully crafted indexes. You may not be aware of this, but just because a column has an index doesn't mean the Query Optimizer will use it. As you can imagine, creating an index that never will be used is a waste of time, and in the worst cases, it can even reduce the performance of your application. Let's learn why.

To start out, let's look at a simple example. Let's assume we have an accounting database. In that database is a table called "orders". Among a number of different columns in this table, we are interested in two columns: "orderid" and "employeeid". This table has 150,000 rows and there is non-clustered index on the "employeeid" table. Now let's say we want to run the following query:

```
SELECT orderid FROM orders WHERE employeeid = 5
```

The first thing to notice about the query is that the "employeeid" column used in the WHERE clause of the query has a non-clustered index on it. Because of this, you would most likely assume that when this query is run through the Query Optimizer, that the Query Optimizer would use the index to produce the requested results.

Unfortunately, you can't automatically make this assumption. Just because there is an available index doesn't necessarily mean that the Query Optimizer will use it. This is because the Query Analyzer always evaluates whether or not an index is useful before it is used. If the Query Analyzer examines an index and finds it not useful, it will ignore it, and if necessary, it will perform a table scan to produce the requested results.

So what is a useful index? In order to answer this question, we need to understand that one of the biggest underlying goals of the Query Optimizer is to reduce the amount of I/O, and the corresponding amount of time it takes to perform execute a query. In other words, the Query Optimizers evaluates many different ways a query can be executed, and finds the one it thinks will produce the least amount of I/O. But what may be surprising is that using an available index does not always mean that it will result in the least amount of I/O used. In many cases, especially with non-clustered indexes, a table scan can produce less I/O than an available index.

Before the Query Optimizer uses an index, the Query Optimizer evaluates the index to see if it is selective enough. What does this mean? Selectivity refers to the percentage of rows in a table that are returned by a query. A query is considered highly selective if it returns a very limited number of rows. A query is considered to have low selectivity if it returns a high percentage of rows. Generally speaking, if a query returns less than 5% of the number of rows in a table, it is considered to have high selectivity, and the index will most likely be used. If the query returns from 5% - 10% of the rows, the index may or may not be used. If the query returns more than 10% of the rows, the index most likely will not be used. And assuming there are no other useful indexes for the query, a table scan will be performed.

Let's return to our example query:

```
SELECT orderid FROM orders WHERE employeeid = 5
```

Just by looking at the query we don't know if the available index on employeeid will be used or not. Let's say that we know that of the 150,000 rows in the table, that "employeeid = 5" is true for 5,000 of the records. If we divide 5,000 by 150,000 we get 3.3%. Since 3.3% is less than 5%, the Query Optimizer will most likely use the available index. But what if "employeeid = 5" is true for 25,000 instead. In this case, we divide 25,000 by 150,000 and we get 16.6%. Since 16.6% is greater than 5%, or even 10%, the Query Optimizer most likely will not use the index and instead perform a table scan.

So how can a table scan use less I/O than using an index, such as the non-clustered index in our example? Non-clustered indexes are great if the index is highly selective, especially if you will be returning one record. But if many records will be returned, and the index is not very selective, it is very expensive in I/O to retrieve the data. The reason for this is that the Query Optimizer has to first go to the index to locate the data (using up I/O) and then it has to go to the table to retrieve it (more I/O). At some point, the Query Optimizer determines that it takes less I/O to scan an entire table than it does to go back and forth between the index and the table to retrieve the requested rows.

The example given above applies mostly to non-clustered indexes. If the available index is a clustered index, then the index may be used, even if there is low selectivity, because the index is the table and I/O operations can be very efficient.

So how does the Query Optimizer know if an available index is selective enough to be useful? It does this by maintaining index statistics on each index in every table. Index statistics are a histogram of values that are stored in the sysindexes table. These statistics are a sampling of the available rows that tells the Query Optimizer approximately how selective a particular index is.

Index statistics are created every time an index is created, rebuilt, when the UPDATE STATISTICS command is run, and automatically by the Query Optimizer as the need arises. Index statistics are not maintained in real time because that would put too much overhead on the server. But because index statistics are not real time, they can get out of date, and sometimes, the Query Optimizer can make a poor choice because the index statistics it uses are not current.

But just because statistics are current doesn't mean that the Query Optimizer will use an available index. Remember, the Query Optimizer bases its decision on the selectivity of an index, and the Query Optimizer uses the index statistics to determine selectivity.

So if the Query Optimizer can check to see if a particular index is useful or not, how can we do the same thing?

Fortunately, there is a command that let's us examine an index and find out if a particular index is selective enough to be used.

The reason we want to know if an index is selective enough or not is because if it isn't, then it won't be used. And if an index won't be used, there is no point in having it. Most likely, dropping unnecessary indexes can boost the performance of your application because indexes, as you probably know, slow down INSERTs, UPDATEs, and DELETEs in a table because of the overhead of maintaining indexes. And if the table in questions is subject to a high level of database changes, index maintenance can be the cause of some bottlenecks. So our goal is to ensure that if we do have indexes, that they are selective enough to be useful. We don't want to maintain indexes that won't be used.

The command we will use to find the selectivity of an index is:

```
DBCC SHOW_STATISTICS (table_name, index_name)
```

When this command runs, it produces an output similar to the following. This is a real result based on one of the databases I maintain.

Statistics for INDEX 'in\_tran\_idx'.

| Updated            | Rows    | Rows Sampled | Steps | Density      | Average key length |
|--------------------|---------|--------------|-------|--------------|--------------------|
| Feb 24 2001 3:36AM | 7380901 | 7163688      | 300   | 2.2528611E-5 | 0.0                |

(1 row(s) affected)

All density Columns

2.2528611E-5 in\_tran\_key

Steps

1  
283  
301  
340  
371  
403  
456  
...  
44510

(300 row(s) affected)

DBCC execution completed. If DBCC printed error messages, contact your system administrator.

This result includes a lot of information, most of which is beyond the scope of this article. What we one to focus on is the density value "2.2528611E-5" under the "All density" column heading.

Density refers to the average percentage of duplicate rows in an index. If an indexed column, such as employeeid, has much duplicate data, then the index is said the have high density. But if an indexed column has mostly unique data, then the index is said to have low density.

Density is inversely related to selectivity. If density is a high number, then selectivity is low, which means an index may not be used. If density is a low number, then selectivity is high, and an index most likely will be used.

In the sample printout above, the density for the index is less than 1%. In turn, this means the selectivity of the table is over 99%, which means that the index is probably very useful for the Query Optimizer.

If you are an advanced DBA, you will probably have already noticed that I have oversimplified this discussion. Even so, the point I want to make in this article is still very valid, and my point is, is that not all indexes are equal. Just because an index is available to the Query Optimizer does not mean it will always be used.

For DBAs, this means you need to be wary of your table's indexes. As time permits, you may want to run the DBCC SHOW\_STATISTICS command and see how selective your indexes actually are. You may find that some of your indexes are not being used. And if this is the case, you may want to consider removing them, which in turn may speed up your

application.

For new DBAs, removing, rather than adding indexes may seem a backward way to performance tune your database. But the more you learn about SQL Server works internally, the better you will understand the limits of using indexes to performance tune your applications.

## Tips on Optimizing Covering SQL Server Indexes

If you have to use a non-clustered index (because your single clustered index can be used better elsewhere in a table), and **if you know that your application will be performing the same query over and over on the same table, consider creating a covering index on the table.** A covering index, which is a form of a composite index, includes all of the columns referenced in SELECT, JOIN, and WHERE clauses of a query. Because of this, the index contains the data you are looking for and SQL Server doesn't have to look up the actual data in the table, reducing logical and/or physical I/O, and boosting performance.

On the other hand, if the covering index gets too big (too many columns), this can increase I/O and degrade performance. Generally, when creating covering indexes, follow these guidelines:

- If the query or queries you run using the covering index are seldom run, then the overhead of the covering index may outweigh the benefits it provides.
- The covering index should not add significantly to the size of the key. If it does, then its use may not outweigh the benefits it provides.
- The covering index must include all columns found in the SELECT list, the JOIN clause, and the WHERE clause.

One clue to whether or not a query can be helped by a covering index is if the execution plan of the query uses a Bookmark Lookup. If it does, then adding a covering index is almost always beneficial.

\*\*\*\*\*

**If a query makes use of aggregates**, and it is run often, then you may want to consider adding a covering index for this query. Non-clustered indexes include a row with an index key value for every row in a table. Because of this, SQL Server can use these entries in the index's leaf level to perform aggregate calculations. This means that SQL Server does not have to go to the actual table to perform the aggregate calculations, which can boost performance.

\*\*\*\*\*

If you want to create a covering index, if possible, **try to piggy-back on already existing indexes** that exist for the table. For example, say you need a covering index for columns c1 and c3. If you already have an index on column c1, instead of creating a new covering index, change the index on c1 to be a composite index on c1 and c3. Anytime you can prevent indexing the same column more than once, the less I/O overhead SQL Server will experience, and the faster performance will be.

\*\*\*\*\*

**How can you tell if a covering index you created is actually being used** by the Query Optimizer? You can find this out by turning on and viewing the graphical execution plan output. If you see this phrase, "Scanning a non-clustered index entirely or only a range", this means that the query optimizer was able to cover that particular query with an index.

\*\*\*\*\*

**An alternative to creating covering indexes** on non-clustered indexes is to let SQL Server create the covering indexes for you automatically. Here's how this works:

The query optimizer can perform what is called index intersection. This allows the optimizer to consider multiple indexes from a table, build a hash table based on the multiple indexes, and then use the hash table to reduce I/O for the query. In effect, the hash table becomes a covering index for the query.

Although index intersection is performed automatically by the query optimizer, you can help it along by creating single column, non-clustered indexes on all the columns in a table that will be queried frequently. This provides the query optimizer with the data it needs to create covering indexes as needed, on the fly.

\*\*\*\*\*

**One way to help determine if a covering index could help a query's performance** is to create a graphical query execution plan in Query Analyzer of the query in question and see if there are any Bookmark Lookups being performed. Essentially, a Bookmark Lookup is telling you that the Query Processor had to look up the row columns it needs from a

table or a clustered index, instead of being able to read it directly from a non-clustered index. Bookmark Lookups can reduce query performance because they produce extra disk I/O to retrieve the column data.

One way to avoid a Bookmark Lookup is to create a covering index. This way, all the columns from the query are available directly from the non-clustered index, which means that Bookmark Lookups are unnecessary, which reduces disk I/O and helps to boost performance.

## Non-clustered indexes are best for queries:

- That return few rows (including just one row) & where the index has good selectivity (generally above 95%).
- That retrieve small ranges of data (not large ranges). Clustered indexes perform better for large range queries.
- Where both the WHERE clause and the ORDER BY clause are both specified for the same column in a query. This way, the non-clustered index pulls double duty. It helps to speed up accessing the records, and it also speeds up the sorting of the records (because the returned data is already sorted).
- That use JOINS (although clustered indexes are better).
- When the column or columns to be indexed are very wide. While wide indexes are never a good thing, but if you have no choice, a non-clustered index will have overall less overhead than clustered index on a wide index.

\*\*\*\*\*

**If a column in a table is not at least 95% unique**, then most likely the query optimizer will not use a non-clustered index based on that column. Because of this, don't add non-clustered indexes to columns that aren't at least 95% unique. For example, a column with "yes" or "no" as the data won't be at least 95% unique.

\*\*\*\*\*

**If your table needs a clustered index, be sure it is added to the table before you add any non-clustered indexes.** If you don't, when you add a clustered index to your table, all of the pre-existing non-clustered indexes will have to be rebuilt (which is done automatically when the clustered index is built), putting an unnecessary strain on your server.

\*\*\*\*\*

**To determine the selectivity on an index on a given table**, run this command: DBCC SHOW\_STATISTICS (table\_name, index\_name). The higher the selectivity of an index, the greater the likelihood it will be used by the query optimizer.

\*\*\*\*\*

**When deciding whether or not to add a non-clustered index to a column of a table, it is useful to first find out how selective it is.** By this, what we want to know is the ratio of unique rows to total rows (based on a specific column) found in the table. Generally, if a column is not more than 95% unique, then the Query Optimizer might not even use the index. If this is the case, then adding the non-clustered index may be a waste of disk space. In fact, adding a non-clustered index that is never used will hurt a table's performance.

Another useful reason to determine the selectivity of a column is to decide what is the best order to position indexes in a composite index. This is because you will get the best performance out of a composite index if the columns are arranged so that the most selective is the first one, the next most selective, the second one, and so on.

So how do you determine the selectivity of a column? One way is to run the following script on any column you are considering for a non-clustered index. This example script is designed to be used with the Northwind database, so you will need to modify it appropriately for your use.

```
--Finds the Degree of Selectivity for a Specific Column in a Row
Declare @total_unique float
Declare @total_rows float
Declare @selectivity_ratio float

SELECT @total_unique = 0
SELECT @total_rows = 0
SELECT @selectivity_ratio = 0

--Finds the Total Number of Unique Rows in a Table
--Be sure to replace OrderID below with the name of your column
--Be sure to replace [Order Details] below with your table name
```

```
SELECT @total_unique = (SELECT COUNT(DISTINCT OrderID) FROM [Order Details])
```

```
--Calculates Total Number of Rows in Table
```

```
--Be sure to replace [Order Details] below with your table name
```

```
SELECT @total_rows = (SELECT COUNT(*) FROM [Order Details])
```

```
--Calculates Selectivity Ratio for a Specific Column
```

```
SELECT @selectivity_ratio = ROUND((SELECT @total_unique/@total_rows),2,2)
```

```
SELECT @selectivity_ratio as 'Selectivity Ratio'
```

The results in this case is 38%, which means that adding a non-clustered index to the OrderID column of the Order Details table in the Northwind database is probably not a very good idea.

\*\*\*\*\*

**In some cases, even though a column (or columns if a composite index) has a non-clustered index, the Query Optimizer may not use it (even though it should), instead performing a table scan (if the table is a heap) or a clustered index scan (if there is a clustered index). This, of course, can produce unwanted performance problems.**

This particular problem can occur when there is a data correlation between the order of the rows in the table, and the order of the non-clustered index entries. This can occur when there is correlation between the clustered index and the non-clustered index. For example, the clustered index may be created on a date column, and the non-clustered index might be created on an invoice number column. If this is the case, then there is a correlation (or direct relationship) between the increasing dates and the increasing invoice numbers found in each row.

The reason this problem occurs is because the Query Optimizer assumes there is no correlation, and it makes its optimization decisions based on this assumption.

If you run into this problem, there are three potential resolutions to this problem:

- If possible, reorder the non-clustered index column (assuming a composite index) so that the column with the highest cardinality is the first column in the composite index.
- Create covering indexes.
- Add index hints to your queries to overrule the Query Optimizer.

\*\*\*\*\*

When you think of page splits, you normally only think of clustered indexes. This is because clustered indexes enforce the physical order of the index, and page splitting can be a problem if the clustered index is based on a non-incrementing column. But what has this to do with non-clustered indexes? While non-clustered indexes use a clustered index (assuming the table is not a heap) as their key, **most people don't realize that non-clustered indexes can suffer from page splitting**, and because of this, need to have an appropriate fillfactor and pad\_index set for them.

Here's an example of how non-clustered indexes can experience page splits. Let's say you have a table that has a clustered index on it, such as customer number. Let's also say that you have a non-clustered index on the zip code column. As you can quite well imagine, the data in the zip code column will have no relation to the customer number and will be more or less random, and data will have to be inserted into the zip code index randomly. Like clustered index pages, non-clustered index pages can experience page splitting.

So just as with clustered indexes, non-clustered indexes need to have an appropriate fillfactor and pad\_index, and also be rebuilt on a periodic basis.

## Tips on Rebuilding SQL Server Indexes

Periodically (daily, weekly, or monthly) perform a **database reorganization** on all the indexes on all the tables in your database. This will rebuild the indexes so that the data is no longer fragmented. Fragmented data can cause SQL Server to perform unnecessary data reads, slowing down SQL Server's performance.

If you do a reorganization on a table with a clustered index, any non-clustered indexes on that same table will automatically be rebuilt.

Database reorganizations can be done using the Maintenance Wizard, or if by running your own custom script via the SQL Server Agent (see below).

DBCC DBREINDEX command will not automatically rebuild all of the indexes on all the tables in a database, it can only work on one table at a time. But if you run the following script, you can index all the tables in a database with ease:

--Script to automatically reindex all tables in a database

```
USE DatabaseName --Enter the name of the database you want to reindex
```

```
DECLARE @TableName varchar(255)
```

```
DECLARE TableCursor CURSOR FOR  
SELECT table_name FROM information_schema.tables  
WHERE table_type = 'base table'
```

```
OPEN TableCursor
```

```
FETCH NEXT FROM TableCursor INTO @TableName  
WHILE @@FETCH_STATUS = 0  
BEGIN  
DBCC DBREINDEX(@TableName, ' ',90)  
FETCH NEXT FROM TableCursor INTO @TableName  
END
```

```
CLOSE TableCursor
```

```
DEALLOCATE TableCursor
```

The script will automatically reindex every index in every table of any database you select, and provide a fillfactor of 90%. You can substitute any number appropriate for the fillfactor in the above script.

When DBCC DBREINDEX is used to rebuild indexes, keep in mind that as the indexes on a specific table are being rebuilt, that the table becomes unavailable for use by your users.

For example, when a non-clustered index is rebuilt, a shared table lock is put on the table, preventing all but SELECT operations to be performed on it. When a clustered index is rebuilt, an exclusive table lock is put on the table, preventing any table access by your users. Because of this, you should only run this command when users don't need access to the tables being reorganized.

\*\*\*\*\*

**When you create or rebuild an index**, you can specify a fillfactor, which is the amount the data pages in the index that are filled. A fillfactor of 100 means that each index page is 100% full, a fillfactor of 50% means each index page is 50% full. If you create a clustered index that has a fillfactor of 100, and it is not based on a monotonically increasing key, that means that each time a record is inserted (or perhaps updated), page splits will occur because there is no room for the data in the existing pages. Numerous page splits can slow down SQL Server's performance.

Here's an example: Assume that you have just created a new index on a table with the default fillfactor. When SQL Server creates the index, it places the index on contiguous physical pages, which allows optimal I/O access because the data can be read sequentially. But as the table grows and changes with INSERTS, UPDATES, and DELETES, page splitting occurs. When pages split, SQL Server must allocate new pages elsewhere on the disk, and these new pages are not contiguous with the original physical pages. Because of this, random I/O, not sequential I/O access must be used to gather the data, which is much slower, to access the index pages.

So what is the ideal fillfactor? It depends on the ratio of reads to writes that your application makes to your SQL Server tables. As a rule of thumb, follow these guidelines:

- Low Update Tables (100-1 read to write ratio): 100% fillfactor
- High Update Tables (where writes exceed reads): 50%-70% fillfactor
- Everything In-Between: 80%-90% fillfactor.

You may have to experiment to find the optimum fillfactor for your particular application. Don't assume that a low fillfactor is always better than a high fillfactor. While page splits will be reduced with a low fillfactor, it also increase the number of pages that have to be read by SQL Server during queries, which reduces performance. And not only is I/O overhead increased with a too low of fillfactor, it also affects your buffer cache. As data pages are moved in from disk to the buffer, the entire page (including empty space) is moved to the buffer. So the lower the fillfactor, the more pages that have to be moved into SQL Server's buffer, which means there is less room for other important data pages to reside at the same time, which can reduce performance.

If you don't specify a fillfactor, the default fillfactor is 0, which means the same as a 100% fillfactor, (the leaf pages of the index are filled 100%, but there is some room left on intermediate index pages). In most cases, this default value is not a good choice, especially for clustered indexes.

\*\*\*\*\*

**If you find that your transaction log grows to an unacceptable size when you run DBCC REINDEX**, you can minimize this growth by switching from the Full Recovery mode to the Bulk-Logged mode before you reindex, and when done, switch back. This will significantly reduce the size of the transaction log growth.

\*\*\*\*\*

**If you have a table that has a clustered index** on a monotonically increasing or decreasing primary key, and if the table is not subject in UPDATES or if it has no VARCHAR columns, then the ideal fillfactor for the table is 100. This is because such a table will normally not experience any page splits. Because of this, there is no point in leaving any room in the index for page splits. And because the fillfactor is 100, SQL Server will require fewer I/Os to read the data in the table, and performance will be boosted.

\*\*\*\*\*

**If you are not sure what to make the fillfactor for your indexes**, your first step is to determine the ratio of disk writes to reads. The way to do this is to use these two counters: Physical Disk Object: % Disk Read Time and Physical Disk Object: % Write Time. When you run both counters on an array, you should get a good feel for what percentage of your I/O is reads and writes. You will want to run this over a period of time representative of your typical server load. If your percentage writes greatly exceeds the percentage of reads, then a lower fillfactor is called for. If your percentage of reads greatly exceeds the percentage of writes, then a higher fillfactor is called for.

Another Performance Monitor counter you can use to help you select the ideal fillfactor for your environment is the SQL Server Access Methods: Pages Splits/Sec. This counter measures the number of page splits that are occurring in SQL Server every second. For best performance, you will want this counter to be as low as possible, as page splits incur extra server overhead, hurting performance. If this number is relatively high, then you may need to lower the fillfactor in order to prevent new page splits. If this counter is very low, then the fillfactor you have is fine, or it could be a little too low. You won't know unless you increase the fillfactor and watch the results.

Ideally, you want a fillfactor that prevents excessive page splits, but not so low as to increase the size of the database, which in turn can reduce read performance because of all the extra data pages that need to be read.

Once you know the ratio of disk write to reads, you now have the information you need to help you determine an optimum fillfactor for your indexes.

\*\*\*\*\*

If you want to **determine the level of fragmentation** of your indexes due to page splitting, you can run the DBCC SHOWCONTIG command. Since this command requires you to know the ID of both the table and index being analyzed, you may want to run the following script:

```

--Script to identify table fragmentation

--Declare variables
DECLARE
@ID int,
@IndexID int,
@IndexName varchar(128)

--Set the table and index to be examined
SELECT @IndexName = 'index_name'           --enter name of index
SET @ID = OBJECT_ID('table_name')         --enter name of table

--Get the Index Values
SELECT @IndexID = IndID
FROM sysindexes
WHERE id = @ID AND name = @IndexName

--Display the fragmentation
DBCC SHOWCONTIG (@id, @IndexID)

```

While the DBCC SHOWCONTIG command provides several measurements, the key one is Scan Density. This figure should be as close to 100% as possible. If the scan density is less than 75%, then you may want to reindex the tables in your database, and also increase the fillfactor if you are finding that the current fillfactor you are using is not appropriate.

\*\*\*\*\*

**Here's a script that is used to create DBCC SHOWCONFIG commands for all of the indexes in one or more tables.** Once you run this script, it will product for you a DBCC SHOWCONFIG statement for each index, which you can then run to find out about the level of fragmentation of your indexes. This script is especially handy if you don't know the names of the indexes in your tables (which is most of the time).

```

SELECT 'dbcc showcontig (' +
CONVERT(varchar(20),i.id) + ',' + -- table id
CONVERT(varchar(20),i.indid) + ') -- ' + -- index id
object_name(i.id) + '.' + -- table name
i.name -- index name
from sysobjects o
inner join sysindexes i
on (o.id = i.id)
where o.type = 'U'
and i.indid < 2
and
i.id = object_id(o.name)
ORDER BY
object_name(i.id), i.indid

```

Once you run this script, the output will be DBCC SHOWCONFIG statements for each of the tables(s) and index(es). This output can then be cut and pasted into Query Analyzer and run, which produces a DBCC SHOWCONFIG result for every index for every table you specified.

\*\*\*\*\*

**Don't reindex your tables when your database is in active production,** as it can lock resources and cause your user's problems. Reindexing should be scheduled during down times, or at the very worst, during very light use of the database.

\*\*\*\*\*

If you use the CREATE INDEX command to create or rebuild your indexes, the FILLFACTOR option has its own sub-option called **PAD\_INDEX**. If you don't specify the PAD\_INDEX option, then the FILLFACTOR only applies to the leaf pages in the index, not the intermediate index pages. But if you specify PAD\_INDEX along the FILLFACTOR option, then when the index is created, the FILLFACTOR percent will be applied to the intermediate index pages.

\*\*\*\*\*

**If you want to rebuild a clustered index using the CREATE INDEX command**, and assuming the table also has non-clustered indexes, the best performance is gained when you also use the DROP\_EXISTING option along with the CREATE INDEX command. The DROP\_EXISTING option includes optimizations that prevent the overhead of rebuilding any of the non-clustered indexes on the table twice.

\*\*\*\*\*

SQL Server 2000 has a command called **DBCC INDEXDEFRAG**, which is used to defrag clustered and non-clustered indexes in a table or indexed view. It does this by defragging and compacting the leaf level of the index so that the physical order of the index pages match the left-to-right logical order of the leaf nodes, which increases performance. Using DBCC INDEXDEFRAG instead of DBCC DBREINDEX is often beneficial because this command does not hold locks for long periods like DBCC DBREINDEX. This means it can be run during production without significantly affecting performance, although running any maintenance task such as this should ideally be scheduled during slow or downtimes.

On the negative side, using DBCC INDEXDEFRAG takes longer to run than DBCC REINDEX, and statistics are not automatically updated. This means that if you use DBCC INDEXDEFRAG, you will also need to run UPDATE STATISTICS.

\*\*\*\*\*

**One way to speed up reindexing your databases** is to be sure that your SQL Server database and log files are physically defragged before you reindex your database. By ensuring that your database and log files are contiguous (defragged), reindexing will not only be faster, but it will require less I/O resources, helping SQL Server's overall performance. If you use Windows 2000 or 2003, a defragging utility is available for this purpose, although the built-in tool will only defrag closed SQL Server database and log files. Ideally, you should use a third-party defragging utility designed to defrag open SQL Server database and log files.

\*\*\*\*\*

According to Microsoft, **the total number of pages in a table affects how page fragmentation affects SQL Server's performance**. For example, if a table has less than 100 data pages, reindexing it to remove fragmentation from it won't benefit performance. This is because other things, such as physical hardware caches, SQL Server caching, and SQL Server read-ahead functionality hides the negative effect of fragmentation. On the other hand, very large table can benefit highly from reindexing because they are so large the fragmentation can negatively affect disk I/O, hurting performance.

## SQL Server Index Tuning Wizard Tips

The **Index Tuning Wizard** is a powerful tool designed to help you identify existing indexes that aren't being used, along with recommending new indexes that can be used to help speed up queries. It uses the actual queries you are running in your database, so its recommendations are based on how your database is really being used. The queries it needs for analysis come from the SQL Server Profiler traces you create.

\*\*\*\*\*

**Use the SQL Server Index Wizard and a appropriate Profiler trace file to help identify potential indexed views** (SQL Server 2000 Enterprise is required). When the Index Wizard runs, it automatically looks for potential indexed views and recommends any that it finds. But don't rely on this tool as the only way to identify indexed views, as it is unable to identify all possible candidates.

\*\*\*\*\*

**If you don't need the full power of the Index Tuning Wizard**, but still want assistance when creating indexes for a table, you can use the "Perform Index Analysis" option located under the "Query" menu of the Query Analyzer in SQL Server 7. For SQL Server 2000, select "Index Tuning Wizard" from the "Query" menu of the Query Analyzer.

Instead of using a Profiler trace file to perform the analysis, it uses the query found in the Query window. While not as comprehensive an analysis as you get from using the Index Tuning Wizard, it is a good starting point when analyzing the performance of specific queries in your database.

\*\*\*\*\*

When creating the Profiler trace that is used to base the Index Tuning Wizard index analysis on, select a **time of day that is representative** of typical transactions run in your SQL Server application. Since the Index tuning wizard bases its recommendations on actual queries, you want the queries in your trace file to be truly representative of how your users use your application.

\*\*\*\*\*

**The longer you run the Profiler trace**, the more accurate the Index Tuning Wizard recommendations will be because you will capture more of the types of typical queries run by your application. But keep in mind that there is an upper limit of 32,767 queries the Index Tuning Wizard can analyze at one time, and that the longer the trace, the longer the analysis will take (could take hours).

\*\*\*\*\*

**If you find that the Index Tuning Wizard takes more time to run** than you have, consider setting the "Maximum columns to per index", under the "Advanced" settings, from the default of 16 columns to a smaller number. This reduces the number of possible indexes the Index Tuning Wizard will evaluate, saving some time, although not a lot. In most cases, especially for OLTP applications, you won't want an index with a lot of columns in it.

\*\*\*\*\*

**Don't capture more in your Profiler trace than you need.** For example, only collect data for a single database, not all of the databases on your server. Also, don't collect more events or data columns that you need to use the Index Wizard. The only events and data columns required by the Index Wizard include the SQL:BatchCompleted and the RPC:Completed events in the TSQL category, and the EventClass and Text data columns. The fewer the events and data columns you capture, the smaller the load Profiler puts on your server when collecting data.

\*\*\*\*\*

**When creating a trace for the Index Tuning Wizard**, consider using the "Create Trace Wizard" to create a "Find the worst performing queries" trace. You can set this trace to only capture queries that run longer than a specified amount of time, such as 1000 milliseconds. Generally, I like to set it at 5 milliseconds, and let it run all day (periodically). Then I use this trace to feed into the Index Tuning Wizard.

My goal here, of course, is to limit the number of queries being tuned to those that are performing the worst. When doing this, be sure you don't have the Index Tuning Wizard evaluate potential indexes to be dropped. This is because you are

not collecting performance data on every query, only the slow ones. If you do evaluate for dropping indexes, you might drop an index that is needed because the queries that use it run faster than the time specified above. [7.0]

\*\*\*\*\*

**Don't run the SQL Server Profiler or the Index Tuning Wizard on your production SQL Servers.** Both tools use SQL Server resources that are best left to your users. Ideally, run them on a workstation connected to the server via your network. In one instance, I was running the Index Wizard on a database with over 800 tables. When the Index Wizard was running, it used over 1GB of virtual memory, greatly slowing my computer. Fortunately, I was using a desktop for the analysis, not my SQL Server. If I had run this same analysis on my production server, my users would have complained loudly.

\*\*\*\*\*

Even if you run the Index Tuning Wizard from a computer other than the one where the database you are analyzing resides, **running the Index Tuning Wizard still puts a load on your production server.** Because of this, you should only run the Index Tuning Wizard when your production database is less busy. Another option is to restore the production to a non-production server, and then run the Index Tuning Wizard against the backup database on the non-production server.

\*\*\*\*\*

Once you have tuned your indexes using the Index Tuning Wizard, don't assume that you are set for life. **Queries often change over time**, and you should periodically rerun the Index Tuning Wizard to see if it recommends any new changes based on the mix of queries that change over time.

\*\*\*\*\*

**Don't blindly accept every recommendation made** by the Index Tuning Wizard. Personally review each recommendation, and based on your knowledge of the database and how it is used, then either accept or not accept the recommendations on a recommendation-by-recommendation basis. For example, the Index Tuning Wizard might recommend adding an index to a table that you know is subject to a tremendous number of INSERTS and UPDATES. Adding an index to such a table may or may not be a good idea.

Also, before blindly taking the Index Tuning wizard's recommendations, review the queries that hit the table that the Index Tuning Wizard is recommending adding an index, and see if perhaps the queries themselves are the problem. Perhaps instead of needing a new index, you really need to rewrite one or more queries.

The Index Tuning Wizard may also recommend you drop one or more indexes. Always carefully review this recommendation before removing any indexes. Remember, the Index Tuning Wizard makes its recommendations based on the trace data you provided it. It is very possible that the trace that was used may not include all relevant data. For example, perhaps you run long reports at night that need certain indexes, and this information was not captured in the trace you created. If you were to delete an index needed for these reports, these reports then may take forever to run because they are missing their needed indexes.

Furthermore, don't rely on the Index Tuning Wizard to recommend all of our table's indexes. You should make the original selection of indexes for your tables based on the types of queries you expect to be run against your data. Only use the Index Tuning Wizard as an adjunct to your original work in order to help fine-tune it.

\*\*\*\*\*

**Sometime, the Index Tuning Wizard will not recommend an index**, even if you know that one is needed. This can happen if the queries are complex, or are part of a larger stored procedure. If you run into this situation, consider breaking up the complex query or stored procedure into smaller queries, and then run these individually through the Index Tuning Wizard.

\*\*\*\*\*

**When the Index Tuning Wizard runs, it creates what are called hypothetical indexes in the sysindexes table.** The names of these indexes start with "hind\_%". These tables are used by the Index Tuning Wizard to help determine if new indexes should be added to your tables.

Normally, these hypothetical tables should be deleted when the Index Wizard is completed, but if the Index Wizard is interrupted before it is completed, it may leave these hypothetical indexes in the sysindexes table.

In some cases, the existence of these tables can lead to an unusual performance problem. What can happen is that some stored procedures may be forced to recompile every time they run, even if they should not be recompiled.

The best way to ensure that you don't have any unnecessary "hind\_%" tables in your sysindexes table is to run a script provided by Microsoft. You can also delete these tables manually from the sysindexes table if you desire. The script can be found here: <http://support.microsoft.com/support/kb/articles/Q293/1/77.ASP>

\*\*\*\*\*

**When you run the Index Tuning Wizard, you have the option to choose if you want a fast, medium, or thorough analysis** (only fast or thorough in 7.0). When analyzing large Profiler traces, your choice makes a significant difference in how fast the analysis is done. But don't try to cut corners here, you should always choose a thorough analysis. After all, you want the optimum indexes for your database, so why would you want to do a less thorough analysis and have a less than optimum set of indexes for your database?

\*\*\*\*\*

In most cases, using the GUI interface for the Index Tuning Wizard is the best way to perform index analysis. But **if you are running a lot of Index Tuning Wizard analysis, you may want to automate this task by using the itwiz command line utility**. You can use this utility, along with the proper command line options, to complete any analysis you want, just as with the GUI interface.

In addition, you can use the SQL Server Agent Scheduler to schedule any analysis jobs to run at convenient times of the day when users will be less affected. See the SQL Server Books Online to learn how to use this command.

\*\*\*\*\*

When using the Index Tuning Wizard to identify potential new indexes, **it ignores triggers on any of your tables when they are executed**. While the Profiler has the ability to capture trigger code using the SQL:StmtCompleted Event, the Index Tuning Wizard is not able to use it for index analysis on trigger code. This means that you must manually tune all indexes used by your trigger manually.

## SQL Server Application and Transact-SQL Performance Checklist

### Performance Audit Checklist

| Transact-SQL Checklist  | Your Response |
|---|---------------|
| Does the Transact-SQL code return more data than needed?  |               |
| Are cursors being used when they don't need to be?  |               |
| Are UNION and UNION SELECT properly used?   |               |
| Is SELECT DISTINCT being used properly?   |               |
| Is the WHERE clause sargable?   |               |
| Are temp tables being used when they don't need to be?  |               |
| Are hints being properly used in queries?   |               |
| Are views unnecessarily being used?   |               |
| Are stored procedures being used whenever possible?   |               |
| Inside stored procedures, is SET NOCOUNT ON being used?   |               |
| Do any of your stored procedures start with sp_?  |               |
| Are all stored procedures owned by DBO, and referred to in the form of database.owner.objectname?   |               |
| Are you using constraints or triggers for referential integrity?  |               |
| Are transactions being kept as short as possible?   |               |
|   |               |
| <b>Application Checklist</b>  |               |
| Is the application using stored procedures, strings of Transact-SQL code, or using an object model, like ADO, to communicate with SQL Server? |               |
| What method is the application using to communicate with SQL Server: DB-LIB, DAO, RDO, ADO, .NET?   |               |
| Is the application using ODBC or OLE DB to communication with SQL Server?   |               |
| Is the application taking advantage of connection pooling?  |               |
| Is the application properly opening, reusing, and closing connections?  |               |
| Is the Transact-SQL code being sent to SQL Server optimized for SQL Server, or is it generic SQL?   |               |
| Does the application return more data from SQL Server than it needs?  |               |
| Does the application keep transactions open when the user is modifying data?  |               |

Enter your results in the table above.

### Application and Transact-SQL Code Greatly Affect SQL Server Performance

Of all the areas that can negatively affect the performance of SQL Server, the application code used to access SQL Server data, including Transact-SQL code, has the biggest potential of hurting performance. Unfortunately though, this is an area that a lot of DBAs don't directly control. And because of this, this area is often neglected when performance tuning SQL Server-based application.

As with previous articles in this series, the purpose of this part of the audit is to catch the "easy" performance-related issues of your application and Transact-SQL code that accesses SQL Server data. Besides the tips listed here, there are a lot more factors that affect SQL Server's performance, but the ones listed here are a good beginning.

Of course, if you are using third-party software, then this part of the performance audit doesn't affect you as you can't do much about the code. But if you have developed your own applications, or if the applications have been developed in-house, then you should be able to take part in this portion of the SQL Server performance audit.

As you review the audit items, and their discussion below, you will quickly discover that identifying some of these issues,

or even fixing them, is no small task. Because of this, it is much better to build your applications with these performance tips in mind instead of having to fix them after the application has been written. You may want keep this article around when building new applications so that you build them for performance the first time around.

## Transact-SQL Checklist

### Does the Transact-SQL Code Return More Data Than Needed?

The less data returned by SQL Server, the less resources SQL Server needs to operate, helping to boost the overall performance of SQL Server. This may sound obvious, but returning unnecessary data is a performance problem that I see over and over.

Here are some of the most common mistakes made by coders when returning data from SQL Server that results in more data than necessary:

- The absence of a WHERE clause. Unless you want to return all data from a table, which is a rare activity, the use of a WHERE clause is necessary to reduce the number of rows returned.
- As an adjunct to the above advice, a WHERE clause needs to be as selective as possible. For example, if you only need to return records from a particular date, don't return all the records for the month, or year. Design the WHERE clause so that exactly only those rows you need returned are returned, and not one extra row.
- In the SELECT clause, only include those columns that you need, not all of them. Along the same line, don't use SELECT \*, as you will most likely be returning more rows than you need.
- I will refer to this one again later on this page, but it also applies here. Don't perform SELECTs against views, instead, bypass the view and get the data you need directly from the table. The reason for this is that many views (of course, not all) return more data than is necessary for the calling SELECT statement, which just ends up returning much more data than necessary.

In case you are not aware of them, here are some of the performance issues caused by returning unnecessary data: Sometimes, returning too much data forces the Query Optimizer to perform a table scan instead of an index lookup; extra I/O is needed to read data; buffer cache space is wasted, which could be better used by SQL Server for other purposes; unnecessary network traffic occurs; on the client, additional data has to be stored in memory which might be better used for other uses; and so on.

### Are Cursors Being Used When They Don't Need to Be?

Cursors of any kind slow SQL Server's performance. While in some cases they cannot be avoided, in many cases they can. So if your application is currently using Transact-SQL cursors, see if the code can be rewritten to avoid them.

If you need to perform row-by-row operations, consider using one or more of these options instead of using a cursor:

- Use temp tables
- Use WHILE loops
- Use derived tables
- Use correlated sub-queries
- Use CASE statements
- Use multiple queries

Each of these above options can substitute for a cursor, and they all perform much faster.

If you can't avoid using cursors, then at least try to speed them up.

### Are UNION and UNION SELECT Properly Used?

Many people don't fully understand how UNION and UNION SELECT work, and because of this, end up wasting a lot of SQL

Server's resources unnecessarily. When UNION is used, it performs the equivalent of a SELECT DISTINCT on the results set. In other words, a UNION will combine two similar recordsets, then search for potential duplicate records and eliminate them. If this is your goal, then using UNION is the correct syntax.

But if the two recordsets you want to combine with the UNION will not have duplicate records, then using UNION is a waste of resources, because it will look for duplicates, even if they don't exist. So if you know there are no duplicates in the recordsets you want to combine, then you will want to use UNION ALL, not UNION. UNION ALL combines recordsets, but does not search for duplicates, which reduces the use of SQL Server resources, boosting overall server performance.

### **Is SELECT DISTINCT Being Used Properly?**

I have noticed that some developers automatically add the DISTINCT clause to the SELECT statement, whether or not it is needed. From a performance perspective, the DISTINCT clause should only be used when its special function is needed, which is to eliminate duplicate records from a recordset. This is because the use of the DISTINCT clause requires that the resultset be sorted and duplicates be eliminated, which uses up valuable SQL Server resources. Of course, if you need to do this, then do so. But if you know that the SELECT statement will never return duplicate records, then using the DISTINCT clause is just an unnecessary waste of SQL Server resources.

### **Is the WHERE Clause Sargable?**

The term "sargable" (which is in effect a made-up word) comes from the pseudo-acronym "SARG", which stands for "Search ARGument," which refers to a WHERE clause that compares a column to a constant value. If a WHERE clause is sargable, this means that it can take advantage of an index to speed completion of the query. If a WHERE clause is non-sargable, this means that the WHERE clause (or at least part of it) cannot take advantage of an index, instead performing a table or index scan, which may cause the query's performance to suffer.

Non-sargable search arguments in the WHERE clause, such as "IS NULL", "<>", "!=", "!>", "!<", "NOT", "NOT EXISTS", "NOT IN", "NOT LIKE", and "LIKE '%500'" generally prevents (but not always) the query optimizer from using an index to perform a search. In addition, expressions that include a function on a column, expressions that have the same column on both sides of the operator, or comparisons against a column (not a constant), are not sargable.

Not every WHERE clause that has a non-sargable expression in it is doomed to a table scan. If the WHERE clause includes both sargable and non-sargable clauses, then at least the sargable clauses can use an index (if one exists) to help access the data quickly.

In many cases, if there is a covering index on the table, which includes all of the columns in the SELECT, JOIN, and WHERE clauses in a query, then the covering index can be used instead of a table scan to return a query's data, even if it has a non-sargable WHERE clause. But keep in mind that covering indexes have their own drawbacks, such as often producing wide indexes that increase disk I/O when they are read.

In some cases, it may be possible to rewrite a non-sargable WHERE clause into one that is sargable. For example, the clause:

```
WHERE SUBSTRING(firstname,1,1) = 'm'
```

can be rewritten like this:

```
WHERE firstname like 'm%'
```

Both of these WHERE clauses produce the same result, but the first one is non-sargable (it uses a function) and will run slow, while the second one is sargable, and will run much faster.

If you don't know if a particular WHERE clause is sargable or non-sargable, check out the query's execution plan in Query Analyzer. Doing this, you can very quickly see if the query will be using indexes or table scans to return your results.

With some careful analysis, and some clever thought, many non-sargable queries can be written so that they are sargable.

### **Are Temp Tables Being Used When They Don't Need to Be?**

While the use of temp tables has many practical uses, like the elimination of a cursor, they still incur overhead, and if that overhead can be eliminated, SQL Server will perform faster. For example, there are a variety of ways that temp tables use can be eliminated, which reduces overhead and boosts performance. Some of the ways to eliminate temp tables include:

- Rewrite the code so that the action you need completed can be done using a standard query or stored procedure.
- Use a derived table.
- Use the SQL Server 2000 "table" datatype. These may, or may not be faster. You must test to be sure.
- Consider using a correlated sub-query.
- Use a permanent table instead.
- Use a UNION statement to mimic a temp table.

Each of these options can often be used to help eliminate temp tables from your Transact-SQL code.

### **Are Hints Being Properly Used in Queries?**

Generally speaking, the SQL Server Query Optimizer does a good job of optimizing queries. But in some rare cases, the Query Optimizer falls down on the job and a query hint is needed to override the Query Optimizer in order to get the best performance out of a query.

While hints can be useful in some situations, they can also be dangerous. Because of this, the use of hints should be done with great care.

One of the biggest issues is inheriting some code that makes a big use of hints, especially code that was written for SQL Server 6.5 or SQL Server 7.0 and is now running under SQL Server 2000. In many cases, hints that were needed under previous versions of SQL Server aren't applicable under newer versions, and their use can actually hurt, not help performance.

In another case, perhaps hints were found to be useful early when an application was first rolled out, but as time passes and the "nature" of the data stored changes over time, once useful hints may no longer apply to the "newer" data, rendering them obsolete and potentially dangerous to performance.

In both of these cases, it is a good idea to periodically reevaluate the benefits of query hints being used. You may find that current hints aren't useful at all, and in fact, hurt performance. And the only way to find this out is to test them in Query Analyzer and see what is actually happening, and then make your decision on whether to continue using them based on what you find out.

### **Are Views Unnecessarily Being Used?**

Views are best used for handling security-related issues, not as a lazy developer's method to store often-used queries. For example, if you need to allow a user adhoc access to SQL Server data, then you might consider creating a view for that user (or group), then giving that user access to the view, and not the underlying tables. On the other hand, from within your application, there is no good reason to SELECT data from views, instead, use Transact-SQL code to SELECT exactly what you want from the tables directly. A view adds unnecessary overhead, and in many cases, causes more data than necessary to be returned, which uses up unnecessary overhead.

For example, let's say that you have a view that returns 10 columns from 2 joined tables. And that you want to retrieve 7 columns from the view with a SELECT statement. What in effect happens is that the query underlying the view runs first, returning data, and then your query runs against the data returned by the query. And since you only need 7 columns, not the 10 columns that are returned by the view, more data than necessary is being returned, wasting SQL Server resources. The rule you should follow in your applications is to always access the base tables directly, not through a view.

### **Are Stored Procedures Being Used Whenever Possible?**

Stored procedures offer many benefits to developers. Some of them include:

- Reduces network traffic and latency, boosting application performance. For example, instead of sending 500 lines of Transact-SQL over the network, all that is needed to be sent over the network is a stored procedure call, which is much faster and uses less resources.
- Stored procedure execution plans can be reused, staying cached in SQL Server's memory, reducing server overhead.

- Client execution requests are more efficient. For example, if an application needs to INSERT a large binary value into an image data column not using a stored procedure, it must convert the binary value to a character string (which doubles its size), and send it to SQL Server. When SQL Server receives it, it then must convert the character value back to the binary format. This is a lot of wasted overhead. A stored procedure eliminates this issue as parameter values stay in the binary format all the way from the application to SQL Server, reducing overhead and boosting performance.
- Stored procedures help promote code reuse. While this does not directly boost an application's performance, it can boost the productivity of developers by reducing the amount of code required, along with reducing debugging time.
- Stored procedures can encapsulate logic. You can change stored procedure code without affecting clients (assuming you keep the parameters the same and don't remove any result sets columns). This saves developer time.
- Stored procedures provide better security to your data. If you use stored procedures exclusively, you can remove direct SELECT, INSERT, UPDATE, and DELETE rights from the tables and force developers to use stored procedures as the method for data access. This saves DBA's time.

As a general rule of thumb, all Transact-SQL code should be called from stored procedures.

### **Inside Stored Procedures, is SET NOCOUNT ON Being Used?**

By default, every time a stored procedure is executed, a message is sent from the server to the client indicating the number of rows that were affected by the stored procedure. Rarely is this information useful to the client. By turning off this default behavior, you can reduce network traffic between the server and the client, helping to boost overall performance of your server and applications.

To turn this feature off on at the stored procedure level, include the statement:

```
SET NOCOUNT ON
```

at the beginning of each stored procedure you write. This statement should be included in every stored procedure you write.

### **Do Any of Your Stored Procedures Start with sp\_?**

If you are creating a stored procedure to run in a database other than the Master database, don't use the prefix "sp\_" in its name. This special prefix is reserved for system stored procedures. Although using this prefix will not prevent a user defined stored procedure from working, what it can do is to slow down its execution ever so slightly.

The reason for this is that by default, any stored procedure executed by SQL Server that begins with the prefix "sp\_", is first attempted to be resolved from within the Master database. Since it is not there, time is wasted looking for the stored procedure.

If SQL Server cannot find the stored procedure in the Master database, then it next tries to resolve the stored procedure name as if the owner of the object is "dbo". Assuming the stored procedure is in the current database, it will then execute. To avoid this unnecessary delay, don't name any of your stored procedures with the prefix "sp\_".

### **Are All Stored Procedures Owned by DBO, and Referred to in the Form of databaseowner.objectname?**

For best performance, all objects that are called from within the same stored procedure should all be owned by the same owner, preferably dbo. If they are not, then SQL Server must perform name resolution on the objects if the object names are the same but the owners are different. When this happens, SQL Server cannot use a stored procedure "in-memory plan" over, instead, it must re-compile the stored procedure, which hinders performance.

When calling a stored procedure from your application, it is also important that you call it using its qualified name. Such as:

```
EXEC dbo.myProcedure
```

instead of:

```
EXEC myProcedure
```

Why? There are a couple of reasons, one of which relates to performance. First, using fully qualified names helps to eliminate any potential confusion about which stored procedure you want to run, helping to prevent bugs and other potential problems. But more importantly, doing so allows SQL Server to access the stored procedures execution plan more directly, and in turn, speeding up the performance of the stored procedure. Yes, the performance boost is very small, but if your server is running tens of thousands or more stored procedures every hour, these little time savings can add up.

### **Are You Using Constraints or Triggers for Referential Integrity?**

Don't implement redundant integrity features in your database. For example, if you are using primary key and foreign key constraints to enforce referential integrity, don't add unnecessary overhead by also adding a trigger that performs the same function. The same goes for using both constraints and defaults or constraints and rules that perform redundant work. While this may sound obvious, it is not uncommon to find these issues in SQL Server databases.

### **Are Transactions Being Kept as Short as Possible?**

Keep all Transact-SQL transactions as short as possible. This helps to reduce the number of locks (of all types), helping to speed up the overall performance of SQL Server. If practical, you may want to break down long transactions into groups of smaller transactions. [More info on how to prevent unnecessary locking.](#)

## **Application Checklist**

### **Is the Application Using Stored Procedures, Strings of Transact-SQL Code, or Using an Object Model, like ADO, to Communicate With SQL Server?**

When an application needs to communicate with SQL Server, essentially it has three choices: it can use stored procedures, strings of Transact-SQL code, or it can use an object model's properties and methods. From a performance perspective, the most efficient is stored procedures, and the least efficient are an object model's properties and methods. Ideally, an application should use stored procedures *only* to access SQL Server.

The advantages of stored procedures have already been described earlier in this article, so they won't be repeated here. A close second are strings of Transact-SQL code sent to SQL Server. If written correctly, query execution plans are automatically reused, helping to boost performance, although you won't get the benefits of a stored procedure, such as reducing network traffic. The problem with using an object model's and methods is that they add an additional layer of code that slows down communications. In addition, often, but not always, the Transact-SQL code created by these properties and methods are not very efficient, further hurting performance.

### **What Method is the Application Using to Communicate With SQL Server: DB-LIB, DAO, RDO, ADO, .NET?**

All applications need to use a data access library (MDAC component) in order to communicate with SQL Server, and there are several to choose from. For best performance, .NET should be your first choice. If you have not yet implemented .NET into your organization, then the next best choice is ADO. Under all conditions, avoid DB-LIB (discontinued but still supported) and DAO, both of which are very slow.

### **Is the Application Using ODBC or OLE DB to Communication with SQL Server?**

If you have a choice between using ODBC or OLE DB to access your SQL Server database, choose OLE DB, as it is generally faster. In addition, using OLE DB allows you to use DSN-less connections, which perform connection maintenance faster than ODBC-based DSN-based connection.

### **Is the Application Taking Advantage of Connection Pooling?**

Try to take advantage of "pooling" to reduce SQL Server connection overhead. Pooling is the term used to refer to the process where a client application can use a preexisting connection from a pool of available connections to connect to SQL Server without having to establish a new connection each time a new connection is required. This reduces SQL Server's overhead and speeds up connections to SQL Server.

Microsoft offers two types of pooling. *Connection pooling* is available through ODBC and can be configured by using the ODBC Data Source Administrator, the registry, or the calling application. *Resource pooling* is available through OLE DB, and can be configured through the application's connection string, the OLE DB API, or the registry.

Either connection pooling or resource pooling can be run for the same connection. Both pooling connections cannot be used for the same connection. Also, for connection pooling to work effectively, so that connections are reused, how security is implemented is critical. For a connection to be reused, the same security context must be used, or another

connection is automatically opened, and connection pooling won't work. Essentially, what this means is that all users who connect from the application to SQL Server must share the same user account. If they don't, then each user will automatically open a new connection when they need to access SQL Server through the application.

For maximum performance, will almost always want to take advantage of one or the other pooling methods when connecting to SQL Server.

### **Is the Application Properly Opening, Reusing, and Closing Connections?**

Generally speaking, a connection to SQL Server should only be opened when it is needed, used, then immediately closed by the application. Assuming that you are using connection pooling and are using the proper security model, what will happen is that if a connection is not currently available, it will be created. And once the connection is closed by the application, it will remain open (although the application thinks that it is closed), available to be reused as needed.

Reducing how often actual connections are opened and closed reduces SQL Server's overhead. Also, by opening and the closing a connection quickly from an application, these allows pooled connections to be more efficiently reused, which also helps to reduce overhead, boosting performance.

### **Is the Transact-SQL Code Being Sent to SQL Server Optimized for SQL Server, or is it Generic SQL?**

Some applications are designed to work with multiple databases, and because of this, use ANSI SQL instead of Transact-SQL to access SQL Server data. While this does make it easier to connect to a wide variety of different flavors of database, it also hurts performance. Transact-SQL offers some specific code that is not available in ANSI SQL which offers performance benefits. Ideally, for best performance, Transact-SQL should be used to access SQL Server, not generic ANSI SQL.

### **Does the Application Return More Data from SQL Server Than it Needs?**

This is similar to one of the audit suggestions for Transact-SQL. Some applications, especially those that allow a user to browse data, return way too much data to the user, often allowing the application to further "limit" the data for the user's benefit. For example, I have seen applications that essentially return all the rows in a table, or a view, to the application, where the application then sorts the data and allows the user to browse through it. If the number of rows is not large, this is OK. But if the number of rows is huge, let's say 100,000 or more, then SQL Server has to produce a huge amount of work (generally a table scan) to return all this data, and the network is also flooded. No user will use all of this data. The application should be designed to only return that data the user really needs at that moment, and not one byte more.

Another example of returning too much data includes applications that allow the user to specify the query criteria. If you must allow users to select their own criteria, it is important to prevent them from returning too many rows by accident. For example, you can use the TOP clause in your SELECT statement, or you can include default parameters for the WHERE clause to prevent users from returning every row in a table.

Returning unneeded data is a terrible waste of resources and a problem that is easily avoided with a little planning.

### **Does the Application Keep Transactions Open When the User is Modifying Data?**

This audit suggestion is also similar to one described above for Transact-SQL. One common activity found in most applications is that a user is allowed to lookup a record, then update it. The key to doing this successfully is to ensure that when you allow a user to do this, that you don't keep the connection open--and the record locked--as it is being updated. If you do, you can create unnecessarily long blocking locks that can hurt SQL Server's performance.

Ideally, from the application's point of view, once the user specifies what record to update, the application should open the connection, select the record, and close the connection. Now the record is on the application's screen. Once the user is done updating it, then the application needs to reopen the connection, update the modified record (assuming it was modified), and then close the connection. It is critical that transactions be kept as short as possible.

## Tips for Reducing SQL Server Locks

**Keep all Transact-SQL transactions as short as possible.** This helps to reduce the number of locks (of all types), helping to speed up the overall performance of your SQL Server applications. If practical, you may want to break down long transactions into groups of smaller transactions. In addition, only include those Transact-SQL commands within a transaction that are necessary for the transaction. Leave all other code outside of the transaction.

\*\*\*\*\*

**An often overlooked cause of locking is an I/O bottleneck.** Whenever your server experiences an I/O bottleneck, the longer it takes user's transactions to complete. And the longer they take to complete, the longer locks must be held, which can lead to other transactions having to wait for previous locks to be released.

If your server is experiencing excessive locking problems, be sure to check if you are also running into an I/O bottleneck. If you do find that you have an I/O bottleneck, then resolving it will help to resolve your locking problem, speeding up the performance of your server.

\*\*\*\*\*

**To help reduce the amount of time tables are locked, which hurts concurrency and performance, avoid interleaving reads and database changes within the same transaction.** Instead, try to do all your reads first, then perform all of the database changes (UPDATES, INSERTS, DELETES) near the end of the transaction. This helps to minimize the amount of time that exclusive locks are held.

\*\*\*\*\*

**Any conditional logic, variable assignment, and other related preliminary setup should be done outside of transactions,** not inside them. Don't ever pause a transaction to wait for user input. User input should always be done outside of a transaction. Otherwise, you will be contributing to locking, hurting SQL Server's overall performance.

\*\*\*\*\*

**Encapsulate all transactions within stored procedures,** including both the BEGIN TRANSACTION and COMMIT TRANSACTION statements in the procedure. This provides two benefits that help to reduce blocking locks.

First, it limits the client application and SQL Server to communications before and after the transaction, thus forcing any messages between the client and the server to occur at a time other than when the transaction is running (reducing transaction time).

Second, It prevents the user from leaving an open transaction (holding locks open) because the stored procedure forces any transactions that it starts to complete or abort.

\*\*\*\*\*

**If you have a client application that needs to "check-out" data** for awhile, then perhaps update it later, or maybe not, you don't want the records locked during the entire time the record is being viewed. Assuming "viewing" the data is much more common than "updating" the data, then one way to handle this particular circumstance is to have the application select the record (not using UPDATE, which will put a share lock on the record) and send it to the client.

If the user just "views" the record and never updates it, then nothing has to be done. But if the user decides to update the record, then the application can perform an UPDATE by adding a WHERE clause that checks to see whether the values in the current data are the same as those that were retrieved.

Similarly, you can check a timestamp column in the record, if it exists. If the data is the same, then the UPDATE can be made. If the record has changed, then the application must include code to notify the user so he or she can decide how to proceed. While this requires extra coding, it reduces locking and can increase overall application performance.

\*\*\*\*\*

**Use the least restrictive transaction isolation level possible** for your user connection, instead of always using the default READ COMMITTED. In order to do this without causing other problems, the nature of the transaction must be carefully analyzed as to what the effect of a different isolation will be.

One example of where not using the default READ COMMITTED isolation level is when running queries to produce reports. In most cases, using an isolation level of READ UNCOMMITTED, will turn off locking, speeding the performance of the query, and other queries hitting the same tables.

\*\*\*\*\*

If your **users are complaining that they have to wait for their transactions to complete**, you may want to find out if object locking on the server is contributing to this problem. To do this, use the SQL Server Locks Object: Average Wait Time (ms). You can use this counter to measure the average wait time of a variety of locks, including: database, extent, Key, Page, RID, and table.

If you identify one or more types of locks causing transaction delays, then you will want to investigate further to see if you can identify what specific transactions are causing the locking. The Profiler is the best tool for this detailed analysis.

\*\*\*\*\*

Use **sp\_who** and **sp\_who2** (the sp\_who2 stored procedure is not documented in the SQL Server Books Online, but offers more detail than sp\_who) to identify which processes may be blocking other processes. While blocking can also be identified using Enterprise Manager, these two commands work much faster and more efficiently.

\*\*\*\*\*

**Try one or more of the following suggestions to help avoid blocking locks:** 1) Use clustered indexes on heavily used tables; 2) Make appropriate use of non-clustered indexes, 3) Try to avoid Transact-SQL statements that affect large numbers of rows at once, especially the INSERT and UPDATE statements; 4) Try to have your UPDATE and DELETE statements use an index; and 5) When using nested transactions, avoid commit and rollback conflicts.

\*\*\*\*\*

**On tables that change little, if at all, such as lookup tables, consider altering the default lock level for the table.** By default, SQL Server uses row level locking for all tables, unless the SQL Query Optimizer determines that a more appropriate locking level, such as page or table locks, is more appropriate. For most lookup tables that aren't huge, SQL Server will automatically use row level locking. Because row locking has to be done at the row level, SQL Server needs to work harder maintain row locks that it does for either page or table locks. Since lookup tables aren't being changed by users, it would be more efficient to use a table lock instead of many individual row locks. How do you accomplish this?

You can override how SQL Server performs locking on a table by using the SP\_INDEXOPTION command. Below is an example of code you can run to tell SQL Server to use page locking, not row locks, for a specific table:

```
SP_INDEXOPTION 'table_name', 'AllowRowLocks', FALSE
GO
SP_INDEXOPTION 'table_name', 'AllowPageLocks', FALSE
GO
```

This code turns off both row and page locking for the table, thus only table locking is available.

\*\*\*\*\*

**If there is a lot of contention for a particular table in your database**, consider turning off page locking for that table, requiring SQL Server to use row level locking instead. This will help to reduce the contention for rows located on the same page. It will also cause SQL Server to work a little harder in order to track all of the row locks. How well this option will work for you depends on the tradeoff in performance between the contention and the extra work SQL Server has to perform. Testing will be needed to determine what is best for your particular environment. Use the SP\_INDEXOPTION stored procedure to turn off page locking for any particular table.

\*\*\*\*\*

**If table scans are used regularly to access data in a table**, and your table doesn't have any useful indexes to prevent this, then consider turning off both row locking and page locking for that table. This in effect tells SQL Server to only use table locking when accessing this table. This will boost access to this table because SQL Server will not have to escalate from row locking, to page locking, to table locking each time a table lock is needed on the table to perform the table scan. On the negative side, doing this can increase contention for the table. Assuming the data in the table is mostly read only,

then this should not be too much of a problem. Testing will be needed to determine what is best for your particular environment.

\*\*\*\*\*

**Do not create temporary tables from within a stored procedure that is invoked by the INSERT INTO EXECUTE statement.** If you do, locks on the syscolumns, sysobjects, and sysindexes tables in the TEMPDB database will be created, blocking others from using the TEMPDB database, which can significantly affect performance.

\*\*\*\*\*

**To help reduce the amount of time it takes to complete a transaction** (and thus reducing how long records are locked) try to avoid using the WHILE statement or Data Definition Language (DDL) within a transaction. In addition, do not open a transaction while browsing data and don't SELECT more data than you need for the transaction at hand. For best performance, you always want to keep transactions as short as possible.

\*\*\*\*\*

**While nesting transactions is perfectly legal, it is not recommended** because of its many pitfalls. If you nest transactions and your code fails to commit or roll back a transaction properly, it can hold locks open indefinitely, significantly impacting performance.

\*\*\*\*\*

**By default in SQL Server, a transaction will wait indefinitely for a lock to be removed** before continuing. If you want, you can assign a locking timeout value to SQL Server so that long running locks won't cause other transactions to wait long periods of time. To assign a locking timeout value to SQL Server, run this command, "SET LOCK\_TIMEOUT length\_of\_time\_in\_milliseconds" from Query Analyzer.

\*\*\*\*\*

**Sometimes you need to perform a mass INSERT or UPDATE** of thousands, if not millions of rows. Depending on what you are doing, this could take some time. Unfortunately, performing such an operation could cause locking problems for your other users. If you know users could be affected by your long-running operation, consider breaking up the job into smaller batches, perhaps even with a WAITFOR statement, in order to allow others to "sneak" in and get some of their work done.

\*\*\*\*\*

In SQL Server 7.0, when a replication snapshot is generated, **SQL Server puts shared locks on all of the tables that are being published for replication.** As you can imagine, this can affect users who are trying to update records in the locked tables. Because of this, you may want to schedule snapshots to be created during less busy times of the day. This is especially true if there are a lot of tables, or if the tables are very large.

In SQL Server 2000, this behavior has changed. Assuming that all subscribers will be either SQL Server 7.0 or 2000 servers, then SQL Server 2000 will use what is called concurrent snapshot processing, which does not put a share lot on the affected tables, helping to boost concurrency.

\*\*\*\*\*

**One way to help reduce locking issues is to identify those transactions that are taking a long time to run.** The longer they take to run, the longer their locks will block other processes, causing contention and reduce performance. The following script can be run to identify current, long-running transactions. This will provide you with a clue as to what transactions are taking a long time, allowing you to investigate and resolve the cause.

```
SELECT spid, cmd, status, loginame, open_tran, datediff(s, last_batch, getdate ()) AS [WaitTime(s)]
FROM master..sysprocesses p
WHERE open_tran > 0
AND spid > 50
AND datediff (s, last_batch, getdate ()) > 30
And EXISTS (SELECT * FROM master..syslockinfo l
WHERE req_spid = p.spid AND rsc_type <> 2)
```

This query provides results based on the instant it runs, and will vary each time you run it. The goal is to review the results and look for transactions that have been open a long time. This generally indicates some problem that should be investigated.

\*\*\*\*\*

**In order to reduce blocking locks in an application, you must first identify them.** Once they are identified, you can then evaluate what is going on, and perhaps be able to take the necessary action to prevent them. The following script can be run to identify processes that have blocking locks that occur longer than a time you specify. You can set the value used to identify blocking locks to any value you want. In the example below, it is for 10 seconds.

```
SELECT spid, waittime, lastwaittype, waitresource
FROM master..sysprocesses
WHERE waittime > 10000 -- The wait time is measured in milliseconds
AND spid > 50 -- Use > 50 for SQL Server 2000, use > 12 for SQL Server 7.0
```

This query measures blocking locks in real-time, which means that only if there is a blocking lock fitting your time criteria when you this query will you get any results. If you like, you can add some additional code that will loop through the above code periodically in order to more easily identify locking issues. Or, you can just run the above code during times when you think that locking is a problem.

## SQL Server Database Job Performance Checklist

### SQL Server Job Performance Audit Checklist

| SQL Server Job Checklist                           | Your Response |
|--|---------------|
| Are you running any unnecessary jobs?              |               |
| Are jobs scheduled to run during production lulls? |               |
| Do any SQL Server jobs on the same server overlap? |               |
| Do you have any non-SQL Server jobs that overlap?  |               |
| Have jobs that run T-SQL been optimized?           |               |
| Have you checked to see how long jobs run?         |               |
| Are there alternative to your current jobs?        |               |

Enter your results in the table above.

### SQL Server Jobs Can Negatively Affect Performance, If You Are Not Careful

Virtually every SQL Server runs one or more daily jobs. And most likely, runs many weekly jobs. Unfortunately, most DBAs set up jobs, and then forget about them, unless of course they break. But if they run day after day without any problems, most jobs are forgotten about.

Just as any application can negatively affect SQL Server's performance, the same is true about jobs. Jobs that run poorly-designed code, or run at bad times, can put a significant strain on SQL Server. Because of this, it is important to include your SQL Servers' jobs as part of your performance audit.

In this section of the SQL Server Performance Audit, we focus on how to identify, and correct, potential job-related performance issues.

#### Are You Running Any Unnecessary Jobs?

Because jobs are often forgotten about, it is very easy to set up a job to accomplish a specific task, and then forget to remove the job when the task is no longer necessary. For example, you may need to create a job that moves data from several tables into another table, nightly, that can be used to produce reports. But if that report is no longer being used, there is no longer any need to run the job, and it should be removed to reduce overhead. The problem is that there is no direct link between the job and the report, so if the report is no longer used, it is easy to forget to remove the job.

As part of your audit, review each of the jobs that are running on each of your servers, and determine if the job is really necessary. If not, then get rid of it.

Along this same line of thinking, look for duplicate jobs. For example, I have seen novice DBAs use the Maintenance Wizard to set up jobs within SQL Server, and not realize exactly what they have done. Then they will add some manual jobs that duplicate one or more of the jobs that were created with the Maintenance Wizard. Doing the same thing twice can contribute to a lot of wasted SQL Server resources.

#### Are Jobs Scheduled to Run During Production Lulls?

As you review each of the jobs on your SQL Servers, take a close look at when they run. Assuming that a job doesn't have to run at a specific time, do your best to schedule jobs so that they run when the SQL Server is less busy, such as at nights or on weekends, depending on your situation.

If you aren't sure when your SQL Server has lulls, do a Performance Monitor log over a week's period. This should provide you with enough data to be able to identify lull periods where you can run non-time-sensitive jobs.

#### Do Any SQL Server Jobs on the Same Server Overlap?

This is a bigger problem than most DBAs realize, especially when a SQL Server has many, many jobs. Just as with any activity on SQL Server, it is ideal if jobs can be spread over time as much as possible, instead of doing all of them at once. For example, if your SQL Server has 10 databases, and you create backup jobs for each of them, it is much better to schedule them to run one at a time, instead of all at the same time.

While you can view how long a job runs from Enterprise Manager, there is no easy way to schedule manual jobs, one after another (giving each job enough time to complete), so that they don't overlap. It can be done, but for servers with lots of jobs, you may need a spreadsheet to figure this all out. As an option, you may want to consider using a third-party tool, such as SQL Sentry, which allows you to view and manage all your jobs visually, ensuring that critical jobs don't overlap.

So as you perform your job audit, check to see that jobs don't overlap, assuming this is possible. If they do overlap, do your best to reschedule them to prevent the overlap, spreading the load over a great a lull time as possible.

### **Do You Have Any Non-SQL Server Jobs that Overlap?**

Besides SQL Server jobs, you may have non-SQL Server jobs on your SQL Server. Some examples of these include defragmentation or tape backup jobs that don't use the SQL Server scheduler. Since these don't use the SQL Server scheduler, they are easy to forget about, and you may end up running some of these jobs at the same time as your SQL Server jobs. As with SQL Server jobs, it is ideal if you can schedule these jobs to run at times other than when your SQL Server jobs run. If need be, include these in the spreadsheet discussed above.

### **Have Jobs that Run T-SQL Been Optimized?**

Just as with code found in applications or scripts, T-SQL that is run as part of a job needs to be optimized. The T-SQL code should follow all the performance tips found on this website, and also any relevant indexes should be added to help the job code run as efficiently as possible.

So for every job that has T-SQL code in it, you should run it through Query Analyzer to views its Execution Plan, looking for potential problems, and also through the Index Wizard, looking for potential useful indexes to boost performance.

### **Have You Checked to See How Long Jobs Run?**

I have already mentioned that you can use Enterprise Manager to view how long any particular job has run. But what I didn't mention is that it is a good idea to check this over time to see if there are a lot of variations in how long a particular job runs. For example, a particular job may normally take 2 minutes to run, but you discover that once a week, on Sundays, that this same job takes over 15 minutes to run. Significant changes in the amount of time that it takes to run a job is a good indication that there is some conflict with this job and another process running on SQL Server. If you find anything like this, you will want to research it in more detail to identify what is going on, and fix it.

### **Are There Alternative to Your Current Jobs?**

Just because you have a job running doesn't mean its the best way to accomplish the task at hand. Evaluate each job, and determine if there is a better way to accomplish the same thing. For example, perhaps writing T-SQL code to perform a nightly import might be more efficient that using your current DTS package. Or perhaps a job you are running, that shells out of SQL Server to run, could better be done from another scheduling program instead of SQL Server. I can't make any specific recommendations here, as there is so much variability involved. But the key thing to remember is that your current jobs are often not the only solution, and that there may be better solutions available--that reduce server overhead--if you take the time to think about them.

## Using Profiler to Identify Poorly Performing Queries

### SQL Server Query Performance Audit Checklist

| SQL Server Job Checklist  | Your Response |
|---|---------------|
| Have you identified all long running queries?                           |               |
| Have you prioritized the queries?                                       |               |
| Have you reviewed the execution plans of the above prioritized queries? |               |

*Enter your results in the table above.*

### Identifying Long Running Queries is First Step

At this step in the SQL Server performance audit, you should have identified all the "easy" performance fixes. Now it is time to get your hands a little dirtier and identify queries (including stored procedures) that run longer than they should, and use up more than their fare share of SQL Server resources.

Slow running queries are ones that take too long to run. So how long is too long? That is a decision you have to make. Generally speaking, I use a cutoff of 5 seconds. In other words, any query running 5 seconds or less is generally fast enough, while queries that take longer than 5 seconds to run are long running. This is an arbitrary decision you have to make. In the company where I work, the report writers, who are the ones who write most of the queries that are run against our databases have a different standard than I have. They only consider a query to be long running if it takes more than 30 seconds to run. So, one of your first steps is to determine what you think a long running query is, and then use this as your standard during this portion of the performance audit.

We don't have unlimited time to tune queries. All we can do is to identify those queries that need the most work, and then work on them. And if we do have time, then we can focus on those queries that are less critical (but still troublesome) to the overall performance of our SQL Servers. Also keep in mind that sometimes, no matter how hard you try to tune a particular query, that there may be little or nothing you can do to improve the performance of a particular query.

### Before You Begin

For this part of the performance audit, you will be using the SQL Profiler tool that comes with SQL Server. As this article focuses on how to perform a performance audit, and not on how to use tools, it is assumed that you know how to use SQL Profiler. If you have not used it before, check out the SQL Server Books Online to get you started on the basics of how to use it.

Before you begin using Profiler to capture the query activity in your SQL Servers, keep the following in mind:

- Don't run the Profiler on the same server you are monitoring, this can noticeably, negatively affect the server's performance. Instead, run it on another server or workstation, and collect the data there.
- When running the Profiler, do not select more data than you need to collect. The more you collect, the more resources are used to collect them, slowing down performance. Only select those events and data columns you really need. I will make recommendation on exactly what to collect shortly.
- Collect data over a "typical" production time, say over a typical 3-4 hour production period. This may vary, depending on how busy your server is. If you don't have a "typical" production time, you may have to collect data over several different periods of a typical production day to get all the data you need.

When you use Profiler, you have two options of how to "set it up." You can choose to use the GUI Profiler interface, or if you like, you can use the built-in Profiler system stored procedures. While using the GUI is somewhat easier, using the stored procedures to collect the data incurs slightly less overhead. In this article, we will be using the GUI interface.

### What Data to Collect

Profiler allows you to specify which events you want to capture and which data columns from those event to capture. In addition, you can use filters to reduce the incoming data to only what you need for this specific analysis. Here's what I

recommend:

## Events to Capture

- Stored Procedures--RPC:Completed
- TSQL--SQL:BatchCompleted

You may be surprised that only two different events need to be captured: one for capturing stored procedures and one for capturing all other Transact-SQL queries.

## Data Columns to Capture

- Duration (data needs to be grouped by duration)
- Event Class
- DatabaseID (If you have more than one database on the server)
- TextData
- CPU
- Writes
- Reads
- StartTime (optional)
- EndTime (optional)
- ApplicationName (optional)
- NTUserName (optional)
- LoginName (optional)
- SPID

The data you want to actually capture and view includes some that are very important to you, especially duration and TextData; and some that are not so important, but can be useful, such as ApplicationName or NTUserName.

## Filters to Use

- Duration > 5000 milliseconds (5 seconds)
- Don't collect system events
- Collect data by individual database ID, not all databases at once
- Others, as appropriate

Filters are used to reduce the amount of data collected, and the more filters you use, the more data you can filter out that is not important. Generally, I use three filters, but others can be used, as appropriate to your situation. And of these, the most important is duration. I only want to collect information on those that have enough duration to be of importance to me, as we have already discussed.

## Collecting the Data

Depending on the filters you used, and the amount of time you run Profiler to collect the data, and how busy your server

is, you may collect a lot of rows of data. While you have several choices, I suggest you configure Profiler to save the data to a file on your local computer (not on the server you are profiling), and not set a maximum file size. Instead, let the file grow as big as it needs to grow. You may want to watch the growth of this file, in case it gets out of hand. In most cases, if you have used appropriate filters, the size should stay manageable. I recommend using one large file because it is easier to identify long running queries if you do.

As mentioned before, collect your trace file during a typical production period, over a period of 3-4 hours or so. As the data is being collected, it will be sorted for you by duration, with the longest running queries appearing at the bottom of the Profiler window. It can be interesting to watch this window for awhile while you are collecting data. If you like, you can configure Profiler to automatically turn itself off at the appropriate time, or you can do this manually.

Once the time is up and the trace stopped, the Profiler trace is now stored in the memory of the local computer, and on disk. Now you are ready to identify those long running queries.

## Analyzing the Data

Guess what, you have already identified all queries that ran during the trace collection that exceed your specified duration, whatever it was. So if you selected a duration of 5 seconds, you will only see those queries that took longer than five seconds to run. By definition, all the queries you have captured need to be tuned. "What! But over 500 queries were captured! That's a lot of work!" It is not as bad as you think. In most cases, many of the queries you have captured are duplicate queries. In other words, you have probably captured the same query over and over again in your trace. So those 500 captured queries may only be 10, or 50, or even 100 distinct queries. On the other hand, there may be only a handful of queries captured (if you are lucky).

Whether you have just a handful, or a lot of slow running queries, your next job is to determine which are the most critical for you to analyze and tune first. This is where you need to set priorities, as you probably don't have enough time to analyze them all.

To prioritize the long running queries, you will probably want to first focus on those that run the longest. But as you do this, keep in mind how often each query is run.

For example, if you know that a particular query is for a report that only runs once a month (and you happened to have captured it when it was running), and this query took 60 seconds to run, it probably is not as high a priority to tune as a query that takes 10 seconds to run, but runs 10 times a minute. In other words, you need to balance the length of how long a query takes to run, to how often it runs. With this in mind, you need to identify and prioritize those queries that take the most physical SQL Server resources to run. Once you have done this, then you are ready to analyze and tune them.

## Analyze Queries by Viewing Their Execution Plans

To analyze the queries that have been captured and prioritized, you will need to move the code to Query Analyzer in order to view the execution plan, so that it can be analyzed. As the focus of this article is on auditing, not analysis, we won't spend the time here to show you how to analyze specific queries. This is a very large subject unto itself, and is covered [here](#).

How you move the code to Query Analyzer for analysis depends on the code. If the code you have captured is Transact-SQL, you can cut and paste it directly into Query Analyzer for analysis. But if the code you have captured is within a stored procedure, you have to do a little more work, because the Profiler does not show what the code in the Stored Procedure looks like, but only shows the name of the stored procedure, along with any parameters that were passed along to it. In this case, to analyze the query in Query Analyzer, you must go to the stored procedure in question, and cut and paste the code to Query Analyzer. Then, assuming there were parameters passed to it, you will have to manually modify the code from the stored procedure so that it will run with the parameters found when it was captured by Profiler.

Now the time-consuming chore begins, and that is the analysis of each query's execution plan to see if there is any way the query can be tuned for better performance. But because you have now identified and prioritized these problematic queries, your time will be much more efficiently spent.

## SQL Server Query Execution Plan Analysis

**When it comes time to analyze the performance of a specific query, one of the best methods is to view the query execution plan.** A query execution plan outlines how the SQL Server query optimizer actually ran (or will run) a specific query. This information is very valuable when it comes time to find out why a specific query is running slow.

There are several different ways to view a query's execution plan. They include:

- From within Query Analyzer is an option called "Show Execution Plan" (located on the Query drop-down menu). If you turn this option on, then whenever you run a query in Query Analyzer, you will get a query execution plan (in graphical format) displayed in a separate window.
- If you want to see an execution plan, but you don't want to run the query, you can choose the option "Display Estimated Execution Plan" (located on the Query drop-down menu). When you select this option, immediately an execution plan (in graphical format) will appear. The difference between these two (if any) is accountable to the fact that when a query is really run (not simulated, as in this option), current operations of the server are also considered. In most cases, plans created by either method will produce similar results.
- When you create a SQL Server Profiler trace, one of the events you can collect is called: MISC: Execution Plan. This information (in text form) shows the execution plan used by the query optimizer to execute the query.
- From within Query Analyzer, you can run the command `SET SHOWPLAN_TEXT ON`. Once you run this command, any query you execute in this Query Analyzer sessions will not be run, but a text-based version of the query plan will be displayed. If the query you are running uses temp tables, then you will have to run the command, `SET STATISTICS PROFILE ON` before running the query.

Of these options, I prefer using the "Show Execution Plan", which produces a graphical output and considers current server operations.

\*\*\*\*\*

**If you see any of the following in an execution plan**, you should consider them warning signs and investigate them for potential performance problems. Each of them are less than ideal from a performance perspective.

- *Index or table scans*: May indicate a need for better or additional indexes.
- *Bookmark Lookups*: Consider changing the current clustered index, consider using a covering index, limit the number of columns in the SELECT statement.
- *Filter*: Remove any functions in the WHERE clause, don't include Views in your Transact-SQL code, may need additional indexes.
- *Sort*: Does the data really need to be sorted? Can an index be used to avoid sorting? Can sorting be done at the client more efficiently?

It is not always possible to avoid these, but the more you can avoid them, the faster your performance will be.

\*\*\*\*\*

**If you have a stored procedure, or other batch Transact-SQL code that uses temp tables, you cannot use the "Display Estimated Execution Plan" option in the Query Analyzer to evaluate it.** Instead, you must actually run the stored procedure or batch code. This is because when a query is run using the "Display Estimated Execution Plan" option, it is not really run, and temp tables are not created. Since they are not created, any references to them in the code will fail, which prevents an estimated execution plan from being created.

On the other hand, if you use a table variable (available in SQL Server 2000) instead of a temp table, you can use the "Display Estimated Execution Plan" option.

\*\*\*\*\*

**If you have a very complex query you are analyzing in Query Analyzer** as a graphical query execution plan, the

resulting plan can be very difficult to view and analyze. You may find it easier to break down the query into its logical components, analyzing each component separately.

\*\*\*\*\*

**The results of a graphical query execution plan are not always easy to read and interpret.** Keep the following in mind when viewing a graphical execution plan:

- In very complex query plans, the plan is divided into many parts, with each part, listed one on top of the other on the screen. Each part represents a separate process or step that the query optimizer had (has) to perform in order to get to the final results.
- Each of the execution plan steps is often broken down into smaller sub-steps. Unfortunately, you don't view the sub-steps from left to right, but from right to left. This means you must scroll to the far right of the graphical query plan to see where each step starts.
- Each of the sub-steps and steps is connected by an arrow, showing the path (order) taken of the query when it was executed.
- Eventually, all of the parts come together at the top left side of the screen.
- If you move your cursor above any of the steps or sub-steps, a pop-up windows is displayed, providing more detailed information about this particular step or sub-step.
- If you move your cursor over any of the arrows connecting the steps and sub-steps, you see a pop-up window showing how many records are being moved from one step or sub-step to another step or sub-step.

\*\*\*\*\*

The arrows that connect one icon to another in a graphical query plan have different thicknesses. **The thickness of the arrow indicates the relative cost in the number of rows and row size of the data moving between each icon.** The thicker the arrow, the more the relative cost is.

You can use this indicator as a quick gauge as to what is happening within the query plan of your query. You will want to pay extra attention to thick arrows in order to see how it affects the performance of your query. For example, thick lines should be at the right of the graphical execution plan, not the left. If you see them on the left, this could indicate that too many rows are being returned, and that the query execution plan is less than optimal.

\*\*\*\*\*

**In an execution plan, each part of it is assigned a percentage cost.** This represents how much this part costs in regard to resource use, relative to the rest of the execution plan. When you analyze an execution plan, you should focus your efforts on those parts that have the largest percentage cost. This way, you focus your limited time on those areas that have the greatest potential for a return on your time investment.

\*\*\*\*\*

**In an execution plan, you may have noticed that some parts of the plan are executed more than once.** As part of your analysis of an execution plan, you should focus some of your time on any part that takes more than one execution, and see if there is any way to reduce the number of executions performed. The fewer executions that are performed, the faster the query will be executed.

\*\*\*\*\*

In an execution plan you will see references to **I/O and CPU cost**. These don't have a "real" meaning, such as representing the use of a specific amount of resources. These figures are used by the Query Optimizer to help it make the best decision. But there is one meaning you can associate with them, and that is that a smaller I/O or CPU cost uses less server resources than a higher I/O or CPU cost.

\*\*\*\*\*

When you examine a graphical SQL Server query execution plan, **one of the more useful thing to look for is how indexes were used (if at all) by the query optimizer to retrieve data from tables from a given query.** By finding

out if an index was used, and how it was used, you can help determine if the current indexes are allowing the query to run as well as it possibly can.

When you place the cursor over a table name (and its icon) in a graphical execution plan, and display the pop-up window, you will see one of several messages. These messages tell you if and how an index was used to retrieve data from a table. They include:

- **Table Scan:** If you see this message, it means there was no clustered index on the table and that no index was used to look up the results. Literally, each row in the table being queried had to be examined. If a table is relatively small, table scans can be very fast, sometimes faster than using an index.

So the first thing you want to do, when you see that a table scan has been performed, is to see how many rows there are in the table. If there are not many, then a table scan may offer the best overall performance. But if this table is large, then a table scan will most likely take a long time to complete, and performance will suffer. In this case, you need to look into adding an appropriate index(s) to the table that the query can use.

Let's say that you have identified a query that uses a table scan, but you also discover that there is an appropriate nonclustered index, but it is not being used. What does that mean, and how come the index was not used? If the amount of data to be retrieved is large, relative to the size of the table, or if the data is not selective (which means that there are many rows with the same values in the same column), a table scan is often performed instead of an index seek because it is faster. For example, if a table has 10,000 rows, and the query returns 1,000 of them, then a table scan of a table with no clustered index will be faster than trying to use a non-clustered index. Or, if the table had 10,000 rows, and 1,000 of the rows have the same value in the same column (the column being used in the WHERE clause), a table scan is also faster than using a non-clustered index.

When you view the pop-up window when you move the cursor over a table in a graphical query plan, notice the "Estimated Row Count" number. This number is the query optimizer's best guess on how many rows will be retrieved. If a table scan was done, and this number is very high, this tells you that the table scan was done because a high number of records were returned, and that the query optimizer believed that it was faster to perform a table scan than use the available non-clustered index.

- **Index Seek:** When you see this, it means that the query optimizer used a non-clustered index on the table to look up the results. Performance is generally very quick, especially when few rows are returned.
- **Clustered Index Seek:** If you see this, this means that the query optimizer was able to use a clustered index on the table to look up the results, and performance is very quick. In fact, this is the fastest type of index lookup SQL Server can do.
- **Clustered Index Scan:** A clustered index scan is like a table scan, except that it is done on a table that has a clustered index. Like a regular table scan, a clustered index scan may indicate a performance problem. Generally, they occur for two different reasons. First, there may be too many rows to retrieve, relative to the total number of rows in the table. See the "Estimated Row Count" to verify this. Second, it may be due to the column queried in the WHERE clause may not be selective enough. In any event, a clustered index is generally faster than a standard table scan, as not all records in the table always have to be searched when a clustered index scan is run, unlike a standard table scan. Generally, the only thing you can do to change a clustered index scan to a clustered index seek is to rewrite the query so that it is more restrictive and fewer rows are returned.

\*\*\*\*\*

**In most cases, the query optimizer will analyze joins and JOIN the tables using the most efficient join type, and in the most efficient order.** But not always. In the graphical query plan, you will see icons that represent the different types of JOINS used in the query. In addition, each of the JOIN icons will have two arrows pointing to it. The upper arrow pointing to the JOIN icon represents the outer table in the join, and the lower arrow pointing to the JOIN icon represent the inner table in the join. Follow the arrows back to see the name of the table being joined.

Sometimes, in queries with multiple JOINS, tracing the arrow back won't reveal a table, but another JOIN. If you place the cursor over the arrows pointing to the upper and lower JOINS, you will see a popup window that tells you how many rows are being sent to the JOIN for processing. The upper arrow should always have fewer rows than the lower arrow. If not, then the JOIN order selected by the query optimizer might be incorrect (see more on this below).

First of all, let's look at JOIN types. SQL Server can JOIN a table using three different techniques: nested loop, hash, and merge. Generally, the fastest type of join is a nested loop, but if that is not feasible, then a hash JOIN or merge JOIN is used (as appropriate), both of which tend to be slower than the nested JOIN.

When very large tables are JOINed, a merge join, not a nested loop join, may be the best option. The only way to really

know is to try both and see which one is the most efficient.

If a particular query is slow, and you suspect it may be because the JOIN type is not the optimum one for your data, you can override the query optimizer's choice by using a JOIN hint. Before you use a JOIN hint, you will want to take some time and learn about each of the JOIN types, and how they are designed to work. This is a complicated subject, beyond the scope of this tip.

JOIN order is also selected by the query optimizer, which it trying to select the most efficient order to JOIN tables. For example, for a nested loop join, the upper table should be the smaller of the two tables. For hash joins, the same is true, the upper table should be the smaller of the two tables. If you feel that the query optimizer is selecting the wrong order, you can override it using JOIN hints.

In many cases, the only way to know for sure if using a JOIN hint to change JOIN type or JOIN order will boost or hinder performance is to give them a try and see what happens.

\*\*\*\*\*

If your SQL Server has multiple CPUs, and you have not changed the default setting in SQL Server to limit SQL Server's ability to use all of the CPUs in the server, then the **query optimizer will consider using parallelism** to execute some queries. Parallelism refers to the ability to execute a query on more than one CPU at the same time. In many cases, a query that runs on multiple processors is faster than a query that only runs on a single processor, but not always.

The Query Optimizer will not always use parallelism, even though it potentially can. This is because the Query Optimizer takes a variety of different things into consideration before it decides to use parallelism. For example, how many active concurrent connections are there, how busy is the CPU, is there enough available memory to run parallel queries, how many rows are being processed, and what is the type of query being run? Once the Query Optimizer collects all the facts, then it decides if parallelism is best for this particular run of the query. You may find that one time a query runs without parallelism, but a few minutes later, the exact same query runs again, but this time, parallelism is used.

In some cases, the overhead of using multiple processors is greater than the resource savings of using them. While the query processor does try to weigh the pros and cons of using a parallel query, it doesn't always guess correctly.

If you suspect that parallelism might be hurting the performance of a particular query, you can turn off parallelism for this particular query by using the OPTION (MAXDOP 1) hint.

The only way to know for sure is to test the query both ways, and see what happens.

\*\*\*\*\*

**When reviewing a graphical execution plan, you may notice that one or more of the icon text is displayed in red**, not in black, as is normal. This means that the related table is missing some statistics that the Query Optimizer would like to have in order to come up with a better execution plan.

To create the missing statistics, right-click on the icon and then select the option, "Create Missing Statistics." This will display the "Create Missing Statistics" dialog box, where you can then easily add the missing statistics.

If you are given the option to update missing statistics, you should always take the opportunity to do so as it will most likely benefit the performance of the query that is being analyzed.

\*\*\*\*\*

**Sometimes, when viewing a graphical query execution plan, you see an icon labeled "Assert."** All this means is that the query optimizer is verifying a referential integrity or check constraint to see if the query will violate it or not. If not, there is no problem. But if it does, then the Query Optimizer will be unable to create an execution plan for the query and an error will be generated.

\*\*\*\*\*

**Often, when viewing a graphical query execution plan, you see an icon labeled "Bookmark Lookup."** Bookmark lookups are quite common to see. Essentially, they are telling you that the Query Processor had to look up the row columns it needs from the table or a clustered index, instead of being able to read it directly from a non-clustered index.

For example, if all of the columns in the SELECT, JOIN, and WHERE clauses of a query don't all exist in the non-clustered

index used to locate the rows that meet the query's criteria, then the Query Optimizer has to do extra work and look at the table or clustered index to find all the columns it needs to satisfy the query.

Another cause of a bookmark lookup is using `SELECT *`, which should never be used.

Bookmark lookups are not ideal from a performance perspective because extra I/O is required to look up all the columns for the rows to be returned.

If you think that a bookmark lookup is hurting a query's performance, you have four potential options to avoid it. First, you can create a clustered index that will be used by the `WHERE` clause, you can take advantage of index intersection, you can create a covering non-clustered index, or you can (if you have SQL Server 2000 Enterprise Edition, create an indexed view. If none of these are possible, or if using one of these will use more resources than using the bookmark lookup, then the bookmark lookup is the optimal choice.

\*\*\*\*\*

**Sometimes, the Query Optimizer will need to create a temporary worktable in the tempdb database.** If this is the case, it will be indicated in the graphical query execution plan with an icon labeled like this: Index Spool, Row Count Spool, or Table Spool.

Anytime that a worktable is used, performance is generally hurt because of the extra I/O generated when maintaining a worktable. Ideally, there should be no worktables. Unfortunately, they cannot always be avoided. And sometimes their use can actually boost performance because using a worktable is more efficient than the alternatives.

In any event, the use of a worktable in a graphical query execution plan should raise an alert with you. Take a careful look at such a query and see if there is anyway it can be rewritten to avoid the work table. There may not be. But if there is, you are one step closer to boosting the performance of the query.

\*\*\*\*\*

**In a graphical query execution plan, often you see the Stream Aggregate icon.** All this means is that some sort of aggregation into a single input is being performed. This is most commonly seen when a `DISTINCT` clause is used, or any aggregation operator, such as `AVG`, `COUNT`, `MAX`, `MIN`, or `SUM`.

\*\*\*\*\*

**The Query Analyzer is not the only tool that can generate and display query execution plans** for queries. The SQL Server Profiler can also display them, albeit in text format only. One of the advantages of using Profiler instead of Query Analyzer to display execution plans is that it can do so for a great many queries from your actual production work, instead of running one at a time using Query Analyzer.

To capture and display query execution plans using Profiler, you must create a trace using the following configuration:

#### *Events to Capture*

- Performance: Execution Plan
- Performance: Show Plan All
- Performance: Show Plan Statistics
- Performance: Show Plan Text

#### *Data Columns to Display*

- StartTime
- Duration
- TextData
- CPU

- Reads
- Writes

### *Filters*

- Duration. You will want to specify a maximum duration, such as 5 seconds, so that you don't get flooded with too much data.

Of course, you can capture more information than is listed above in your trace, the above is only a guideline. But keep in mind that you don't want to capture too much data, as this could have a negative affect on your server's performance as the trace is being run.

\*\*\*\*\*

**If you use the OPTION FAST hint in a query**, be aware that the Execution Plan results may not be what you expect. The Execution Plan that you get is based on the results of using the FAST hint, not the actual Execution Plan for the full query.

The FAST hint is used to tell the Query Optimizer to return the specified number of rows as fast as possible, even if they hurts the overall performance of the query. The purpose of this hint is to return a specified number of records quickly in order to produce an illusion of speed for the user. Once the specified number of rows is returned, the remaining rows are returned as they would be normally.

So if you are using the FAST hint, the execution plan will be for only those rows that are returned FAST, not for all of the rows. If you want to see the execution plan for all the rows, then you must perform an Execution Plan of the query with the hint removed.

## **How to Best Implement a SQL Server Performance Audit**

### **The Final Word on How to Perform a SQL Server Performance Audit**

If you have gotten this far, you have done a lot of reading. In this final article on how to perform a SQL Server Audit, we will take a look at some best practices about how to best implement a SQL Server Performance Audit. You will want to read this before you begin any actual performance audits on your SQL Servers.

### **Develop Your Own Custom Performance Audit**

At this point, I assume you have read, or at least skimmed through all of the suggested audit steps. And I imagine that you may have read some things that don't really apply to you. This only makes sense since most SQL Server installations are somewhat different. Because of this, I recommend that you customize this audit for your particular circumstances, adding or deleting steps that better meet your needs.

### **Use Microsoft Word or Excel to Maintain Your Audit Checklist**

As you perform audits on each of your SQL Servers, you will need a way to track the results. While you have lots of options, cutting the applicable checklists from this article series and pasting them into a Word or Excel document is a quick way to get started. You will probably want to create a separate checklist for every server. If you decide to use Excel for your audit worksheet, you can enter all of the checklist items on their own row, and then create a separate column for each server being audited. This way, you can quickly view the results of each of your SQL Servers.

### **Prioritize Your SQL Servers and Databases**

If you manage a lot of SQL Servers and databases, you may not know where to start the performance audit. Ideally, you should prioritize your SQL Servers, and their databases, into the ones that need the most performance help now, and others that may not need as much help. This will help you determine where to start. Most likely, you won't be able to audit them all at once. Instead, audit them when you can, in the order from most important to least important.

### **Keep the Focus of This Performance Audit in Mind**

As you perform the audit on your SQL Server, keep in mind that the goal is to identify and fix the easy problems. But, as you can imagine, you will probably also identify some more difficult to resolve issues. To keep you sane, and to help you better manage your limited time, you will want focus on the easy ones now, and save the hard ones until all of the easy ones are taken care of first. So as you perform the audit and identify issues, prioritize them into easy and difficult categories, and save the difficult ones for when you have time to focus on them.

### **Don't Jump the Gun**

As you perform the audit, you will be tempted to make fixes and changes as you run across them. In most cases, doing so probably shouldn't be a problem. But ideally, it is better to first perform the audit, then based what you found, decide on a formal approach to resolving the issues you identified, and then implement them in a methodical manner.

### **A Recommended Step, But Perhaps Too Much to Ask**

In a perfect world, with lots of time, it would be a good idea to perform a performance benchmark on your server, perform the audit, make any needed changes, and then perform another performance benchmark to see what happened. This will let you know immediately if what you did was helpful or not, and in some cases, not the right thing to do. While this suggestion is highly recommended, it may not be practical from a time perspective. But if you do have the time, then you should seriously consider it.

### **Another Recommended Step, But Perhaps Also Too Much to Ask**

After performing an audit, you may find that just a change or two on a single SQL Server is all that is needed, but on others, perhaps dozens of changes need to be made. If there are a lot of changes to be made, it is probably a wise choice to not implement them all at once, but one at a time, or several at a time. This way, you can see the effect of each change, or set of changes, makes to the server. If you were to make many changes at once, and then experience a problem, you wouldn't know which change caused the problem, requiring you to undo all of the changes, and then try them one by one until you discover the culprit.

### **This Recommendation is Not to Much to Ask**

If the server you need to make changes to is a mission-critical production server, you will want to be very careful about any changes you make. Ideally, you should test any changes on a test SQL Server before implementing them on a production server. If this is not practical, then make only one change at a time, and be sure you know how to reverse the change should there be any problems. In addition, try to pick a less busy time of the day to make the changes, in case there are problems.

### **Have a Backout Plan**

Most of the changes you will be making because of the audit will be easy to reverse. But some may not be so easy. In those cases, you need to have a backout plan just in case you need it. For example, backup your system and user databases before you make any critical changes. That way, if there should be a problem, then you can restore your server to the state it was in before you made the change. I don't want to scare you away from making changes, but you should always be prepared.

### **Document All Changes**

As you make changes based on your performance audit, be sure that you document all changes. This way, if there are any problems later, it will be much easier to identify what went wrong. Probably, the easiest way to document your changes is to add them to your audit spreadsheet, or other document you used to collect the audit information.

### **Perform SQL Server Performance Audits Yearly**

Over time, many SQL Servers (but not all) change. Settings change, service packs are added, and even data changes. All of these can affect performance. The best way to ensure optimum performance on your SQL Servers is do an annual performance audit.

### **After I Complete an Audit and Make the Changes, What Do I Do Next?**

Take it easy? Nope. Just the opposite. Remember, this audit is designed to catch the obvious and easy to correct SQL Server performance issues. Once you have done this, next, you need to identify and correct the hard to correct problems. The performance audit, as previously mentioned, may have identified some difficult problems, and others you may have to discover as they occur. In any event, you will mostly likely be spending much more time identifying and correcting the hard problems that you did on the initial performance audit. But like anything else, focus on those problems that cause the biggest performance problems, and then work your way through them as time permits. Good luck!