SQL Server

Microsoft SQL Server 2000 RDBMS Performance Tuning Guide for Data Warehousing

John H. Miller and Henry Lau

Microsoft Corporation

June 2001

Summary: Provides database administrators and developers with valuable information on Microsoft® SQL Server™ 2000 performance and tuning concepts, with specific information for the business intelligence developer. (92 printed pages)

Audience

This performance tuning guide is designed to help database administrators and developers configure Microsoft® SQL Server™ 2000 for maximum performance and to assist in determining causes of poor performance of relational databases, including those used in data warehousing. It also provides guidelines and best practices for loading, indexing, and writing queries to access data stored in SQL Server. Various SQL Server tools that can be used to analyze performance characteristics are also discussed.

SQL Server 2000 performance and tuning philosophy

Microsoft SQL Server 7.0 introduced a major enhancement: a database engine that is largely self-configuring, self-tuning, and self-managing. Before SQL Server 7.0, most database servers required a considerable amount of time and effort from the database administrator, who had to manually tune the server configuration to achieve optimal performance. In fact, a good many competitive database offerings still require administrators to manually configure and tune their database server. This is a key reason many customers are turning to SQL Server. SQL Server 2000 builds upon the solid foundation laid by SQL Server 7.0. The goal of SQL Server is to make manual configuration and tuning of a database server an obsolete and archaic practice.

By reducing the amount of time required to configure and tune the database environment, SQL Server 2000 enables customers to redirect their efforts toward more productive endeavors. Readers familiar with the earlier version of this document, "MS SQL Server 7.0 Performance Tuning Guide," will notice that fewer options in SQL Server 2000 need to be manually adjusted in order to achieve good performance.

While it is still possible to manually configure and adjust some **sp_configure** options, it is recommended that database administrators refrain from doing so and instead allow SQL Server to automatically configure and tune itself. SQL Server 7.0 has an established and proven track record for being able to make such adjustments; SQL Server 2000 significantly improves on this time-proven formula. Letting SQL Server self-tune allows the database server to dynamically adjust to changing conditions in your environment that could have an adverse effect on database performance.

Basic Principles of Performance Tuning

You can take a number of actions to manage the performance of your databases. SQL Server 2000 provides several tools to assist you in these tasks.

Managing Performance

- Let SQL Server do most of the tuning.

  SQL Server 2000 has been dramatically enhanced to create a largely auto-configuring and self-tuning database server. Take advantage of SQL Server's auto-tuning settings to help SQL Server run at peak performance even as user load and queries change over time.

- Manage RAM caching.

  RAM is a limited resource. A major part of any database server environment is the management of random access memory (RAM) buffer cache. Access to data in RAM cache is much faster than access to the same information from disk. But RAM is a limited resource. If database I/O (input/output operations to the physical disk subsystem) can be reduced to the minimal required set of data and index pages, these pages will stay in RAM longer. Too much unneeded data and index information flowing into buffer cache will quickly push out valuable pages. The primary goal of performance tuning is to reduce I/O so that buffer cache is best utilized.

- Create and maintain good indexes.

  A key factor in maintaining minimum I/O for all database queries is ensuring that good indexes are created and maintained.

- Partition large data sets and indexes.

  To reduce overall I/O contention and improve parallel operations, consider partitioning table data and indexes. Multiple techniques for achieving and managing partitions using SQL Server 2000 are addressed in this document.

- Monitor disk I/O subsystem performance.

The physical disk subsystem must provide a database server with sufficient I/O processing power for the database server to run without disk queuing. Disk queuing results in bad performance. This document describes how to detect disk I/O problems and how to resolve them.

- Tune applications and queries.

  This becomes especially important when a database server will be servicing requests from hundreds or thousands of connections through a given application. Because applications typically determine the SQL queries that will be executed on a database server, it is very important for application developers to understand SQL Server architectural basics and how to take full advantage of SQL Server indexes to minimize I/O.

- Optimize active data.

  In many business intelligence databases, a significant majority of database activity involves data for the most recent month or quarter — as much as 80 percent of database activity may be due to the most recently loaded data. To maintain good overall database performance, make sure this data gets loaded, indexed, and partitioned in a way that provides optimal data access performance for it.

Take Advantage of SQL Server Performance Tools

- SQL Profiler and the Index Tuning Wizard

  SQL Profiler can be used to monitor and log the workload of a SQL Server. This logged workload can then be submitted to the SQL Server Index Tuning Wizard so index changes can be made to help performance if necessary. SQL Profiler and Index Tuning Wizard help administrators achieve optimal indexing. Using these tools periodically will keep SQL Server performing well, even if the query workload changes over time.

- SQL Query Analyzer and Graphical Execution Plan

  In SQL Server 2000, Query Analyzer provides Graphical Execution Plan, an easy method for analyzing problematic SQL queries. Statistics I/O is another important feature of SQL Query Analyzer described later in this document.

- System Monitor objects

  SQL Server includes a complete set of System Monitor objects and counters to provide information for monitoring and analyzing the operations of SQL Server. This document describes key counters to watch.

Configuration Options That Impact Performance

max async IO

A manual configuration option in SQL Server 7.0, **max async IO** has been automated in SQL Server 2000. Previously, **max async IO** was used to specify the number of simultaneous disk I/O requests that SQL Server 7.0 could submit to Microsoft Windows® 2000 and Windows NT® 4.0 during a checkpoint operation. In turn, Windows submitted these requests to the physical disk subsystem. The automation of this configuration setting enables SQL Server 2000 to automatically and dynamically maintain optimal I/O throughput.

Note Windows 98 does not support asynchronous I/O, so the max async IO option is not supported on this platform.

Database Recovery Models

SQL Server 2000 introduces the ability to configure how transactions are logged at a database level. The model chosen can have a dramatic impact on performance, especially during data loads. There are three recovery models: Full, Bulk-Logged, and Simple. The recovery model of a new database is inherited from the **model** database when the new database is created. The model for a database can be changed after the database has been created.

- Full Recovery provides the most flexibility for recovering databases to an earlier point in time.
- Bulk-Logged Recovery provides higher performance and lower log space consumption for certain large-scale operations (for example, create index or bulk copy). It does this at the expense of some flexibility of point-in-time recovery.
- Simple Recovery provides the highest performance and lowest log space consumption, but it does so with significant exposure to data loss in the event of a system failure. When using the Simple Recovery model, data is recoverable only to the last (most recent) full database or differential backup. Transaction log backups are not usable for recovering transactions because, in this model, the transactions are truncated from the log upon checkpoint. This creates the potential for data loss. After the log space is no longer needed for recovery from server failure (active transactions), it is truncated and reused.

Knowledgeable administrators can use this recovery model feature to significantly speed up data loads and bulk operations. However, the amount of exposure to data loss varies with the model chosen.

Important It is imperative that the risks be thoroughly understood before choosing a recovery model.

Each recovery model addresses a different need. Trade-offs are made depending on the model you chose. The trade-offs that occur pertain to performance, space utilization (disk or tape), and protection against data loss. When you

choose a recovery model, you are deciding among the following business requirements:

- Performance of large-scale operations (for example, index creation or bulk loads)
- Data loss exposure (for example, the loss of committed transactions)
- Transaction log space consumption
- Simplicity of backup and recovery procedures

Depending on what operations you are performing, one model may be more appropriate than another. Before choosing a recovery model, consider the impact it will have. The following table provides helpful information.

Recovery model	Benefits	Work loss exposure	Recover to point in time?
Simple	Permits high-performance bulk copy operations. Reclaims log space to keep space requirements small.	Changes since the most recent database or differential backup must be redone.	Can recover to the end of any backup. Then changes must be redone.
Full	No work is lost due to a lost or damaged data file. Can recover to an arbitrary point in time (for example, prior to application or user error).	Normally none. If the log is damaged, changes since the most recent log backup must be redone.	Can recover to any point in time.
Bulk-Logged	Permits high-performance bulk copy operations. Minimal log space is used by bulk operations.	If the log is damaged, or bulk operations occurred since the most recent log backup, changes since that last backup must be redone. Otherwise, no work is lost.	Can recover to the end of any backup. Then changes must be redone.

Multi-Instance Considerations

SQL Server 2000 also introduces the ability to run multiple instances of SQL Server on a single computer. By default, each instance of SQL Server dynamically acquires and frees memory to adjust for changes in the workload of the instance. Performance tuning can be complicated when multiple instances of SQL Server 2000 are each automatically and independently adjusting memory usage. This feature is not generally a consideration for most high-end business intelligence customers who typically install only a single instance of SQL Server on each computer. However, as individual machines become significantly larger (Windows 2000 Datacenter Server supports up to 64 gigabytes (GB) RAM and 32 CPUs), the desire for multiple instances may come into play even in some production environments. Special considerations apply to instances that utilize extended memory support.

Extended Memory Support

Generally speaking, because SQL Server 2000 dynamically acquires and frees memory as needed, it is not usually necessary for an administrator to specify how much memory should be allocated to SQL Server. However, SQL Server 2000 Enterprise Edition and SQL Server 2000 Developer Edition introduce support for using Microsoft Windows 2000 Address Windowing Extensions (AWE). This enables SQL Server 2000 to address significantly more memory (approximate maximum of 8 GB for Windows 2000 Advanced Server and 64 GB for Windows 2000 Datacenter Server). When extended memory is configured, each instance accessing the extended memory must be configured to statically allocate the memory it will use.

Note This feature is available only if you are running Windows 2000 Advanced Server or Windows 2000 Datacenter Server.

Windows 2000 Usage Considerations

To take advantage of AWE memory, you must run the SQL Server 2000 database engine under a Windows 2000 account that has been assigned the Windows 2000 lock pages in memory privilege. SQL Server Setup will automatically grant the MSSQLServer service account permission to use the **Lock Page in Memory** option. If you are starting an instance of SQL Server 2000 from the command prompt using Sqlservr.exe, you must manually assign this permission to the interactive user's account using the Windows 2000 Group Policy utility (Gpedit.msc), or SQL Server will be unable to use AWE memory when not running as a service.

To enable the Lock Page in Memory option

- On the **Start** menu, click **Run**, and then in the **Open** box, enter **gpedit.msc**.
- In the **Group Policy** tree pane, expand **Computer Configuration**, and then expand **Windows Settings**.
- Expand **Security Settings**, and then expand **Local Policies**.

- Select the **Users Rights Assignment** folder.
- The policies will be displayed in the details pane.
- In the details pane, double-click **Lock pages in memory**.
- In the **Local Security Policy Setting** dialog box, click **Add**.
- In the **Select Users or Groups** dialog box, add an account with privileges to run **Sqlservr.exe**.

To enable Windows 2000 Advanced Server or Windows 2000 Datacenter Server to support more than 4 GB of physical memory, you must add the */pae* parameter to the Boot.ini file.

For computers with 16 GB or less you can use the */3gb* parameter in the Boot.ini file. This enables Windows 2000 Advanced Server and Windows 2000 Datacenter Server to allow user applications to address extended memory through the 3 GB of virtual memory, and it reserves 1 GB of virtual memory for the operating system itself.

If more than 16 GB of physical memory is available on a computer, the Windows 2000 operating system needs 2 GB of virtual memory address space for system purposes. Therefore, it can support only a 2 GB virtual address space for application usage. For systems with more than 16 GB of physical memory, be sure to use the */2gb* parameter in the Boot.ini file.

Note If you accidentally use the /3gb parameter, Windows 2000 will be unable to address any memory above 16 GB.

SQL Server 2000 Usage Considerations

To enable the use of AWE memory by an instance of SQL Server 2000, use **sp_configure** to set the **awe enabled** option. Next, restart SQL Server to activate AWE. Because AWE support is enabled during SQL Server startup and continues until SQL Server is shut down, SQL Server will notify users when AWE is in use by sending an "Address Windowing Extension enabled" message to the SQL Server error log.

When you enable AWE memory, instances of SQL Server 2000 do not dynamically manage the size of the address space. Therefore, when you enable AWE memory and start an instance of SQL Server 2000, one of the following occurs, depending on how you have set **max server memory**.

- If **max server memory** has been set and there are at least 3 GB of free memory available on the computer, the instance acquires the amount of memory specified in **max server memory**. If the amount of memory available on the computer is less than **max server memory** (but more than 3 GB), then the instance acquires almost all of the available memory and may leave only up to 128 MB of memory free.
- If **max server memory** has not been set and there is at least 3 GB of free memory available on the computer, the instance acquires almost all of the available memory and may leave only up to 128 MB of memory free.
- If there is less than 3 GB of free memory available on the computer, memory is dynamically allocated and, regardless of the parameter setting for **awe enabled**, SQL Server will run in nonAWE mode.

When allocating SQL Server AWE memory on a 32-GB system, Windows 2000 may require at least 1 GB of available memory to manage AWE. Therefore, when starting an instance of SQL Server with AWE enabled, it is recommend you do not use the default **max server memory** setting, but instead limit it to 31 GB or less.

Failover Clustering and Multi-Instance Considerations

If you are using SQL Server 2000 failover clustering or running multiple instances while using AWE memory, you must ensure that the summed value of the **max server memory** settings for all running SQL Server instances is less than the amount of physical RAM available. For failover, you have to take into consideration the lowest amount of physical RAM on any candidate surviving node. If a failover node has less physical memory than the original node, the instances of SQL Server 2000 may fail to start or may start with less memory than they had on the original node.

sp_configure Options

cost threshold for parallelism Option

Use the **cost threshold for parallelism** option to specify the threshold where SQL Server creates and executes parallel plans. SQL Server creates and executes a parallel plan for a query only when the estimated cost to execute a serial plan for the same query is higher than the value set in **cost threshold for parallelism**. The cost refers to an estimated elapsed time in seconds required to execute the serial plan on a specific hardware configuration. Only set **cost threshold for parallelism** on symmetric multiprocessors (SMP).

Longer queries usually benefit from parallel plans; the performance advantage negates the additional time required to initialize, synchronize, and terminate the plan. The **cost threshold for parallelism** option is actively used when a mix of short and longer queries is executed. The short queries execute serial plans while the longer queries use parallel plans. The value of **cost threshold for parallelism** determines which queries are considered short, thus executing only serial plans.

In certain cases, a parallel plan may be chosen even though the query's cost plan is less than the current **cost threshold for parallelism** value. This is because the decision to use a parallel or serial plan, with respect to **cost threshold for parallelism**, is based on a cost estimate provided before the full optimization is complete.

The **cost threshold for parallelism** option can be set to any value from 0 through 32767. The default value is 5 (measured in milliseconds). If your computer has only one processor, if only a single CPU is available to SQL Server because of the value of the **affinity mask** configuration option, or if the **max degree of parallelism** option is set to 1, SQL Server ignores **cost threshold for parallelism**.

max degree of parallelism Option

Use the **max degree of parallelism** option to limit the number of processors (a maximum of 32) to use in parallel plan execution. The default value is 0, which uses the actual number of available CPUs. Set the **max degree of parallelism** option to 1 to suppress parallel plan generation. Set the value to a number greater than 1 to restrict the maximum number of processors used by a single query execution. If a value greater than the number of available CPUs is specified, the actual number of available CPUs is used.

Note If the **affinity mask** option is not set to the default, the number of CPUs available to SQL Server on symmetric multiprocessor (SMP) systems may be restricted.

For servers running on an SMP computer, change **max degree of parallelism** rarely. If your computer has only one processor, the **max degree of parallelism** value is ignored.

priority boost Option

Use the **priority boost** option to specify whether SQL Server should run at a higher scheduling priority than other processes on the same computer. If you set this option to one, SQL Server runs at a priority base of 13 in the Windows scheduler. The default is 0, which is a priority base of seven. The **priority boost** option should be used only on a computer dedicated to SQL Server, and with an SMP configuration.

Caution Boosting the priority too high may drain resources from essential operating system and network functions, resulting in problems shutting down SQL Server or using other Windows tasks on the server.

In some circumstances, setting **priority boost** to anything other than the default can cause the following communication error to be logged in the SQL Server error log:

```
Error: 17824, Severity: 10, State: 0 Unable to write to ListenOn
connection '<servername>', loginname '<login ID>', hostname '<hostname>'
OS Error: 64, The specified network name is no longer available.
```

Error 17824 indicates that SQL Server encountered connection problems while attempting to write to a client. These communication problems may be caused by network problems, if the client has stopped responding, or if the client has been restarted. However, error 17824 does not necessarily indicate a network problem and may simply be a result of having the **priority boost** option set to on.

set working set size Option

Use the **set working set size** option to reserve physical memory space for SQL Server that is equal to the server memory setting. The server memory setting is configured automatically by SQL Server based on workload and available resources. It will vary dynamically between **min server memory** and **max server memory**. Setting **set working set size** means the operating system will not attempt to swap out SQL Server pages even if they can be used more readily by another process when SQL Server is idle.

Do not set **set working set size** if you are allowing SQL Server to use memory dynamically. Before setting **set working set size** to 1, set both **min server memory** and **max server memory** to the same value, the amount of memory you want SQL Server to use.

The options **lightweight pooling** and **affinity mask** are discussed in the section "Key Performance Counters to Watch" later in this document.

Optimizing Disk I/O Performance

When configuring a SQL Server that will contain only a few GB of data and not sustain heavy read or write activity, it is not as important to be concerned with the subject of disk I/O and balancing of SQL Server I/O activity across hard drives for maximum performance. But to build larger SQL Server databases that will contain hundreds of gigabytes or even terabytes of data and/or that can sustain heavy read/write activity, it is necessary to drive configuration around maximizing SQL Server disk I/O performance by load-balancing across multiple hard drives.

Optimizing Transfer Rates

One of the most important aspects of database performance tuning is I/O performance tuning. SQL Server is certainly no exception. Unless SQL Server is running on a machine with enough RAM to hold the entire database, I/O performance will be determined by how fast reads and writes of SQL Server data can be processed by the disk I/O subsystem.

Because transfer rates, I/O throughput, and other factors which may impact I/O performance are constantly improving, we will not provide specific numbers on what kinds of speed you should expect to see from your storage system. To better understand the capabilities you can expect, it is recommended that you work with your preferred hardware vendor to determine the optimum performance to expect.

What we do want to emphasize is the difference between sequential I/O operations (also commonly referred to as "serial" or "in disk order") in contrast to nonsequential I/O operations. We also want to draw attention to the dramatic effect read-ahead processing can have on I/O operations.

Sequential and Nonsequential Disk I/O Operations

It is worthwhile to explain what these terms mean in relation to a disk drive. Generally, a single hard drive consists of a set of drive platters. Each platter provides surfaces for read/write operations. A set of arms with read/write heads is used to move across the platters and read/write data from/to the platter surfaces. With respect to SQL Server, these are the two important points to remember about hard drives.

First, the read/write heads and associated disk arms need to move in order to locate and operate on the location of the hard drive platter that SQL Server requests. If the data is distributed around the hard drive platter in nonsequential locations, it takes significantly more time for the hard drive to move the disk arm (seek time) and to spin the read/write heads (rotational latency) to locate the data. This contrasts with the sequential case, in which all of the required data is co-located on one contiguous physical section of the hard drive platter, so the disk arm and read/write heads move a minimal amount to perform the necessary disk I/O. The time difference between the nonsequential and the sequential case is significant: about 50 milliseconds for each nonsequential seek in contrast to approximately two to three milliseconds for sequential seeks. Note that these times are rough estimations and will vary based upon how far apart the nonsequential data is spread around on the disk, how fast the hard disk platters can spin (RPM), and other physical attributes of the hard drive. The main point is, sequential I/O is good for performance and nonsequential I/O is detrimental to performance.

Second, it is important to remember that it takes almost as much time to read or write 8 kilobytes (KB) as it does to read or write 64 KB. Within the range of 8 KB to about 64 KB it remains true that disk arm plus read/write head movement (seek time and rotational latency) account for the majority of the time spent for a single disk I/O transfer operation. So, mathematically speaking, it is beneficial to try to perform 64-KB disk transfers as often as possible when more than 64 KB of SQL Server data needs to be transferred, because a 64-KB transfer is essentially as fast as an 8-KB transfer and eight times the amount of SQL Server data is processed for each transfer. Remember that read-ahead manager does its disk operations in 64-KB chunks (referred to as a SQL Server extent). The log manager performs sequential writes in larger I/O sizes, as well. The main point to remember is that making good use of the read-ahead manager and separating SQL Server log files from other nonsequentially accessed files benefit SQL Server performance.

As a rule of thumb, most hard drives can deliver performance that is as much as 2 times better when processing sequential I/O operations as compared to processing nonsequential I/O operations. That is, operations that require nonsequential I/O take twice as long to carry out as sequential I/O operations. What this tells us is that, if possible, you should avoid situations that may lead to random I/O occurring within your database. While it should always be the goal to perform I/O operations sequentially, situations like page splitting or out of sequence data do tend to cause nonsequential I/O to occur.

To encourage sequential I/O it is important to avoid situations that cause page splitting. It is also helpful to devise a well thought out data loading strategy. You can encourage data to be laid out sequentially on disk by employing a partitioning strategy that separates data and indexes. It is important that you set up jobs to periodically check for fragmentation in your data and indexes, and that you use utilities provided with SQL Server to resequence the data when it becomes too fragmented. More information about doing these operations appears later in this document.

Note Logs generally are not a major concern because transaction log data is always written sequentially to the log file in sizes ranging up to 32 KB.

RAID

RAID (redundant array of inexpensive disks) is a storage technology often used for databases larger than a few gigabytes. RAID can provide both performance and fault tolerance benefits. A variety of RAID controllers and disk configurations offer tradeoffs among cost, performance, and fault tolerance. This topic provides a basic introduction to using RAID technology with SQL Server databases and discusses various configurations and tradeoffs.

- **Performance.** Hardware RAID controllers divide read/writes of all data from Windows NT 4.0 and Windows 2000 and applications (like SQL Server) into slices (usually 16–128 KB) that are then spread across all disks participating in the RAID array. Splitting data across physical drives like this has the effect of distributing the read/write I/O workload evenly across all physical hard drives participating in the RAID array. This increases disk I/O performance because the hard disks participating in the RAID array, as a whole are kept equally busy, instead of some disks becoming a bottleneck due to uneven distribution of the I/O requests.
- **Fault tolerance.** RAID also provides protection from hard disk failure and accompanying data loss by using two methods: mirroring and parity.

Mirroring is implemented by writing information onto a second (mirrored) set of drives. If there is a drive loss with mirroring in place, the data for the lost drive can be rebuilt by replacing the failed drive and rebuilding the mirrorset. Most RAID controllers provide the ability to do this failed drive replacement and remirroring while Windows and SQL Server are online. Such RAID systems are commonly referred to as "Hot Plug" capable drives.

One advantage of mirroring is that it offers the best performance among RAID options if fault tolerance is required.

Bear in mind that each SQL Server write to the mirrorset results in two disk I/O operations, once to each side of the mirrorset. Another advantage is that mirroring provides more fault tolerance than parity RAID implementations. Mirroring can enable the system to survive at least one failed drive and may be able to support the system through failure of up to half of the drives in the mirrorset without forcing the system administrator to shut down the server and recover from the file backup.

The disadvantage of mirroring is cost. The disk cost of mirroring is one extra drive for each drive worth of data. This essentially doubles your storage cost, which, for a data warehouse, is often one of the most expensive components needed. Both RAID 1 and its hybrid, RAID 0+1 (sometimes referred to as RAID 10 or 0/1) are implemented through mirroring.

Parity is implemented by calculating recovery information about data written to disk and writing this parity information on the other drives that form the RAID array. If a drive should fail, a new drive is inserted into the RAID array and the data on that failed drive is recovered by taking the recovery information (parity) written on the other drives and using this information to regenerate the data from the failed drive. RAID 5 and its hybrids are implemented through parity. The advantage of parity is cost. To protect any number of drives with RAID 5, only one additional drive is required. Parity information is evenly distributed among all drives participating in the RAID 5 array.

The disadvantages of parity are performance and fault tolerance. Due to the additional costs associated with calculating and writing parity, RAID 5 requires four disk I/O operations for each write, compared to two disk I/O operations for mirroring. Read I/O operation costs are the same for mirroring and parity. Read operations, however, are usually one failed drive before the array must be taken offline and recovery from backup media must be performed to restore data.

General Rule of Thumb: Be sure to stripe across as many disks as necessary to achieve solid disk I/O performance. System Monitor will indicate if there is a disk I/O bottleneck on a particular RAID array. Be ready to add disks and redistribute data across RAID arrays and/or small computer system interface (SCSI) channels as necessary to balance disk I/O and maximize performance.

Effect of On-Board Cache of Hardware RAID Controllers

Many hardware RAID controllers have some form of read and/or write caching. This available caching with SQL Server can significantly enhance the effective I/O handling capacity of the disk subsystem. The principle of these controller-based caching mechanisms is to gather smaller and potentially nonsequential I/O requests coming in from the host server (SQL Server) and try to batch them together with other I/O requests for a few milliseconds so that the batched I/Os can form larger (32–128 KB) and maybe sequential I/O requests to send to the hard drives. In keeping with the principle that sequential and larger I/O is good for performance, this helps produce more disk I/O throughput given the fixed number of I/Os that hard disks are able to provide to the RAID controller. It is not that the RAID controller caching magically allows the hard disks to process more I/Os per second. Rather, the RAID controller cache is using some organization to arrange incoming I/O requests to make best possible use of the underlying hard disks' fixed amount of I/O processing ability.

These RAID controllers usually protect their caching mechanism with some form of backup power. This backup power can help preserve the data written in cache for some period of time (perhaps days) in case of a power outage. If the database server is also supported by an uninterruptible power supply (UPS), the RAID controller has more time and opportunity to flush data to disk in the event of power disruption. Although a UPS for the server does not directly affect performance, it does provide protection for the performance improvement supplied by RAID controller caching.

RAID Levels

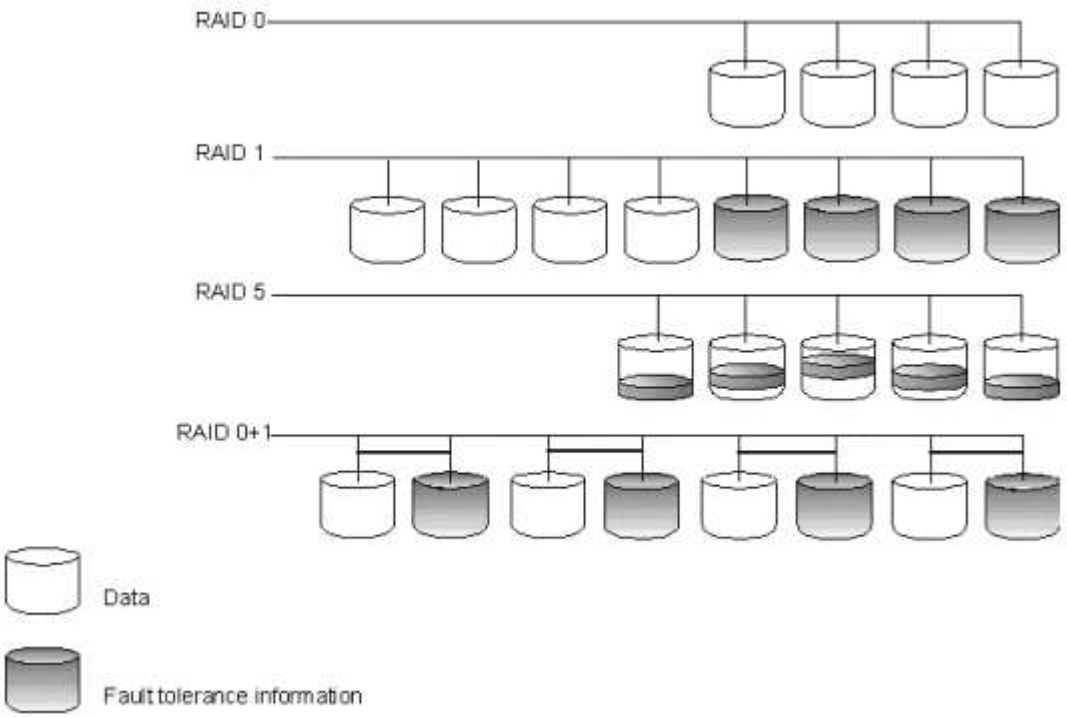
As mentioned above, RAID 1 and RAID 0+1 offer the best data protection and best performance among RAID levels, but cost more in terms of disks required. When cost of hard disks is not a limiting factor, RAID 1 or RAID 0+1 are the best choices in terms of both performance and fault tolerance.

RAID 5 costs less than RAID 1 or RAID 0+1 but provides less fault tolerance and less write performance. The write performance of RAID 5 is only about half that of RAID 1 or RAID 0+1 because of the additional I/O needed to read and write parity information.

The best disk I/O performance is achieved with RAID 0 (disk striping with no fault tolerance protection). Because RAID 0 provides no fault tolerance protection, it should never be used in a production environment, and it is not recommended for development environments. RAID 0 is typically used only for benchmarking or testing.

Many RAID array controllers provide the option of RAID 0+1 (also referred to as RAID 1/0 and RAID 10) over physical hard drives. RAID 0+1 is a hybrid RAID solution. On the lower level, it mirrors all data just like normal RAID 1. On the upper level, the controller stripes data across all of the drives (like RAID 0). Thus, RAID 0+1 provides maximum protection (mirroring) with high performance (striping). These striping and mirroring operations are transparent to Windows and SQL Server because they are managed by the RAID controller. The difference between RAID 1 and RAID 0+1 is on the hardware controller level. RAID 1 and RAID 0+1 require the same number of drives for a given amount of storage. For more information on RAID 0+1 implementation of specific RAID controllers, contact the hardware vendor that produced the controller.

The illustration below shows differences between RAID 0, RAID 1, RAID 5, and RAID 0+1.

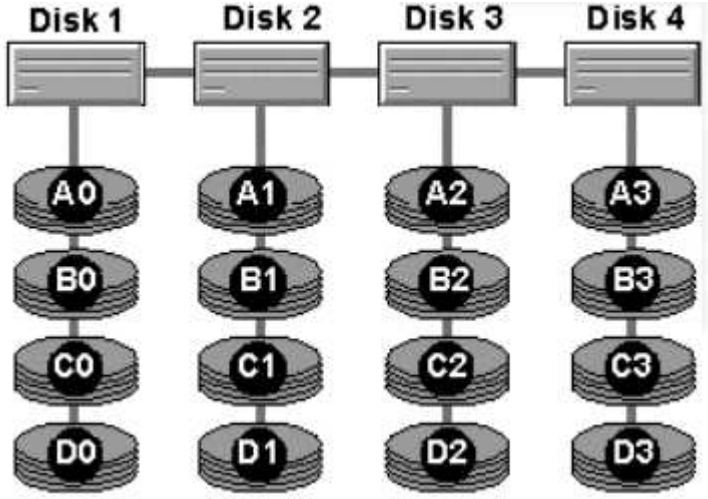


If your browser does not support inline frames, click here to view on a separate page.

Note In the illustration above, in order to hold four disks worth of data, RAID 1 (and RAID 0+1) need eight disks, whereas Raid 5 only requires five disks. Be sure to involve your storage vendor to learn more about their specific RAID implementation.

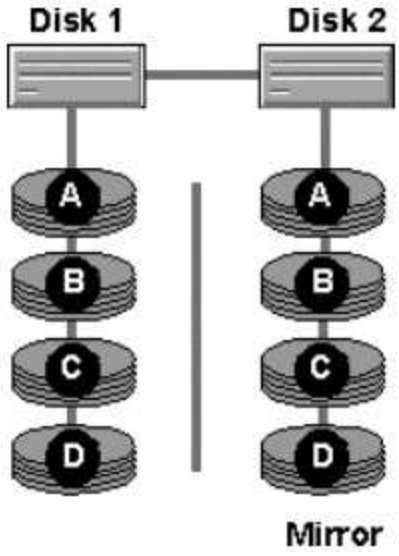
Level 0

This level is also known as disk striping because of its use of a disk file system called a stripe set. Data is divided into blocks and spread in a fixed order among all disks in an array. RAID 0 improves read/write performance by spreading operations across multiple disks, so that operations can be performed independently and simultaneously. RAID 0 is similar to RAID 5, except RAID 5 also provides fault tolerance. The following illustration shows RAID 0.



Level 1

This level is also known as disk mirroring because it uses a disk file system called a mirror set. Disk mirroring provides a redundant, identical copy of a selected disk. All data written to the primary disk is written to the mirror disk. RAID 1 provides fault tolerance and generally improves read performance (but may degrade write performance). The following illustration shows RAID 1.



Level 2

This level adds redundancy by using an error correction method that spreads parity across all disks. It also employs a disk-striping strategy that breaks a file into bytes and spreads it across multiple disks. This strategy offers only a marginal improvement in disk utilization and read/write performance over mirroring (RAID 1). RAID 2 is not as efficient as other RAID levels and is not generally used.

Level 3

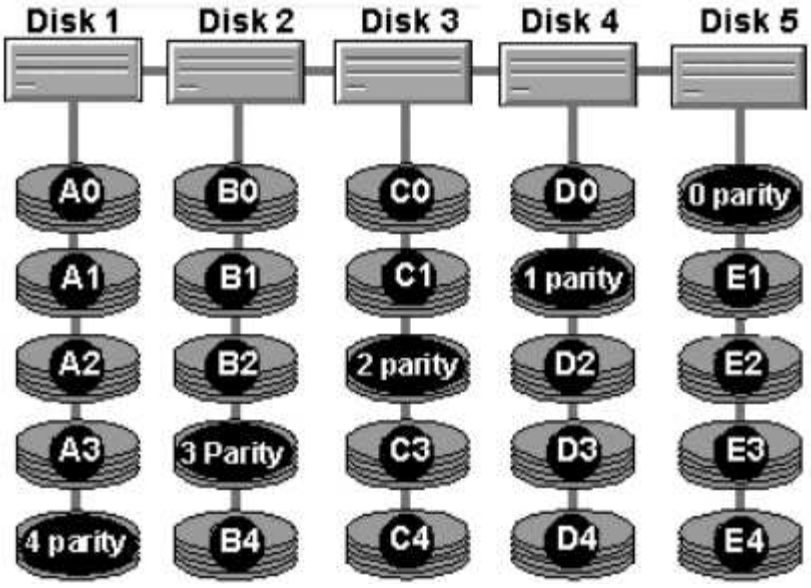
This level uses the same striping method as RAID 2, but the error correction method requires only one disk for parity data. Use of disk space varies with the number of data disks. RAID 3 provides some read/write performance improvement. RAID 3 also is rarely used.

Level 4

This level employs striped data in much larger blocks or segments than RAID 2 or RAID 3. Like RAID 3, the error correction method requires only one disk for parity data. It keeps user data separate from error-correction data. RAID 4 is not as efficient as other RAID levels and is not generally used.

Level 5

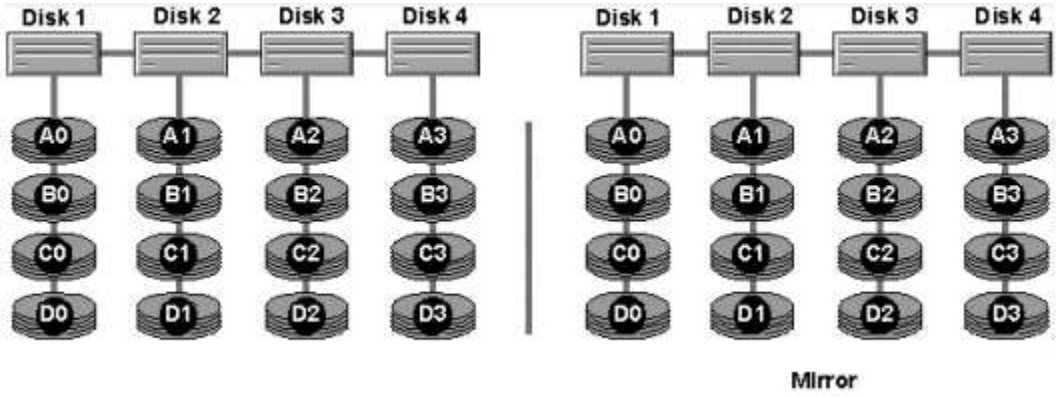
Also known as striping with parity, this level is the most popular strategy for new designs. It is similar to RAID 4 because it stripes the data in large blocks across the disks in an array. It differs in how it writes the parity across all the disks. Data redundancy is provided by the parity information. The data and parity information are arranged on the disk array so the two are always on different disks. Striping with parity offers better performance than disk mirroring (RAID 1). However, when a stripe member is missing, read performance degrades (for example, when a disk fails). RAID 5 is one of the most commonly used RAID configurations. The following illustration shows RAID 5.



If your browser does not support inline frames, click here to view on a separate page.

Level 10 (1+0)

This level is also known as mirroring with striping. This level uses a striped array of disks, which are then mirrored to another identical set of striped disks. For example, a striped array can be created using four disks. The striped array of disks is then mirrored using another set of four striped disks. RAID 10 provides the performance benefits of disk striping with the disk redundancy of mirroring. RAID 10 provides the highest read/write performance of any of the RAID levels at the expense of using twice as many disks. The following illustration shows RAID 10.



If your browser does not support inline frames, click here to view on a separate page.

Online RAID Expansion

This feature allows disks to be added dynamically to a physical RAID array while SQL Server remains online. Additional disk drives are automatically integrated into the RAID storage. Disk drives are added by installing them into physical positions called hot plug drive slots, or hot plug slots. Many hardware vendors offer hardware RAID controllers that are capable of providing this functionality. Data is automatically re-striped across all drives evenly, including the newly added drive, and there is no need to shut down SQL Server or Windows. You can take advantage of this functionality by leaving hot plug slots free in the disk array cages. If SQL Server is regularly overtaxing a RAID array with I/O requests (this will be indicated by Disk Queue Length for the Windows logical drive letter associated with that RAID array), it is possible to install one or more new hard drives into the hot plug slots while SQL Server is still running. The RAID controller will move some existing SQL Server data to these new drives so data is evenly distributed across all drives in the RAID array. Then the I/O processing capacity of the new drives (75 nonsequential/150 sequential I/Os per second, for each drive) is added to the overall I/O processing capacity of the RAID array.

System Monitor and RAID

In System Monitor (Performance Monitor in Microsoft Windows NT® 4.0), information can be obtained for both logical and physical disk drives. The difference is that logical disks in System Monitor are associated with what Windows reads as a logical drive letter. Physical disks in System Monitor are associated with what Windows reads as a single physical

hard disk.

In Windows NT 4.0, all disk counters for Performance Monitor were turned off by default because they could have a minor impact on performance. In Windows 2000 the physical disk counters are turned on by default and the logical disk counters are turned off by default. **Diskperf.exe** is the Windows command that controls the types of counters that can be viewed in System Monitor.

In Windows 2000, to obtain performance counter data for logical drives or storage volumes, you must type `diskperf -yv` at the command prompt, and then press ENTER. This causes the disk performance statistics driver used for collecting disk performance data to report data for logical drives or storage volumes. By default, the operating system uses the `diskperf -yd` command to obtain physical drive data.

The syntax for **Diskperf.exe** in Windows 2000 is as follows:

```
diskperf [-y[d|v] | -n[d|v]] [\\computername]
```

Parameters

(none)

Reports whether disk performance counters are enabled and identifies the counters enabled.

-y

Sets the system to start all disk performance counters when you restart the computer.

-yd

Enables the disk performance counters for physical drives when you restart the computer.

-yv

Enables the disk performance counters for logical drives or storage volumes when you restart the computer.

-n

Sets the system to disable all disk performance counters when you restart the computer.

-nd

Disables the disk performance counters for physical drives.

-nv

Disables the disk performance counters for logical drives.

computername

Specifies the computer you want to see or set disk performance counters to use.

With Windows NT 4.0 and earlier, `diskperf -y` was used for monitoring hard drives, or sets of hard drives and RAID controllers, that were not using Windows NT software RAID. When utilizing Windows software RAID, use `diskperf -ye` so that System Monitor will report physical counters across the Windows NT stripesets correctly. When `diskperf -ye` is used in conjunction with Windows NT stripesets, logical counters will not report correct information and should be disregarded. If logical disk counter information is required in conjunction with Windows NT stripesets, use `diskperf -y` instead. With `diskperf -y`, logical disk counters will be reported correctly for Windows NT stripesets, but physical disk counters will not report correct information and should be disregarded.

Note The effects of the diskperf command do not take effect until Windows has been restarted (both for Windows 2000 and earlier versions of Windows NT).

Considerations for Monitoring Hardware RAID

Because RAID controllers present multiple physical hard drives as a single RAID mirrorset or stripeset to Windows, Windows reads the grouping as though it were a single physical disk. The resulting abstracted view of the actual

underlying hard drive activity can cause performance counters to report information that can be misleading.

From a performance tuning perspective, it is very important to be aware of how many physical hard drives are associated with a RAID array. This information will be needed when determining the number of disk I/O requests that Windows and SQL Server are sending to each physical hard drive. Divide the number of disk I/O requests that System Monitor reports as being associated with a hard drive by the number of actual physical hard drives known to be in that RAID array.

To get a rough estimate of I/O activity for each hard drive in a RAID array, it is also important to multiply the number of disk write I/Os reported by System Monitor by either two (RAID 1 and 0+1) or four (RAID 5). This will give a more accurate account of the number of actual I/O requests being sent to the physical hard drives, because it is at this physical level that the I/O capacity numbers for hard drives apply. This method, however, will not calculate the hard drive I/O exactly, when the hardware RAID controller is using caching, because caching can significantly affect the direct I/O to the hard drives.

When monitoring disk activity, it is best to concentrate on disk queuing instead of on the actual I/O for each disk. Disk I/O speeds depend on the transfer rate capability of the drives, which cannot be adjusted. Because there is little you can do other than buy faster, or more, drives, there is little reason to be concerned with the amount of I/O that is actually occurring. However, you do want to avoid too much disk queuing. Significant disk queuing reveals that you have an I/O problem. Because Windows cannot read the number of physical drives in a RAID array, it is difficult to accurately assess disk queuing for each physical disk. A rough approximation can be determined by dividing the Disk Queue Length by the number of physical drives participating in the hardware RAID disk array for the logical drive being observed. It is optimal to attempt to keep the disk queue number below two for hard drives containing SQL Server files.

Software RAID

Windows 2000 supports software RAID to address fault tolerance by providing mirrorsets and stripesets (with or without fault tolerance) through the operating system when a hardware RAID controller is not used. You can set up RAID 0, RAID 1, or RAID 5 functionality using operating system procedures. Most large data warehouses use hardware RAID, but in the event that your installation is relatively small or you choose not to implement hardware RAID, software RAID can provide some data access and fault tolerance advantages.

Software RAID does utilize some CPU resources, because Windows has to manage the RAID operations that the hardware RAID controller would typically manage for you. Thus, performance with the same number of disk drives and Windows software RAID may be a few percent less than with hardware RAID, especially if the system processors are nearly 100 percent utilized for other purposes. By reducing the potential for I/O bottlenecks, Windows software RAID will generally help a set of drives service SQL Server I/O better than if the drives are used without software RAID. Software RAID should allow for better CPU utilization by SQL Server because the server will wait less often for I/O requests to complete.

Disk I/O Parallelism

An effective technique for improving the performance of large SQL Server databases that are stored on multiple disk drives is to create disk I/O parallelism, which is the simultaneous reading from and writing to multiple disk drives. RAID implements disk I/O parallelism through hardware and software. The next topic discusses using partitioning to organize SQL Server data to further increase disk I/O parallelism.

Partitioning for Performance

For SQL Server databases that are stored on multiple disk drives, performance can be improved by partitioning the data to increase the amount of disk I/O parallelism.

Partitioning can be done using a variety of techniques. Methods for creating and managing partitions include configuring your storage subsystem (disk, RAID partitioning) and applying various data configuration mechanisms in SQL Server such as files, filegroups, tables and views. While this section focuses on some of the partitioning capabilities as they relate to performance, the white paper titled "Using Partitions in a SQL Server 2000 Data Warehouse" specifically addresses the subject of partitioning.

The simplest technique for creating disk I/O parallelism is to use hardware partitioning and create a single "pool of drives" that serves all SQL Server database files except transaction log files, which should always be stored on physically separate disk drives dedicated to log files only. The pool may be a single RAID array that is represented in Windows as a single physical drive. Larger pools may be set up using multiple RAID arrays and SQL Server files/filegroups. A SQL Server file can be associated with each RAID array and the files can be combined into a SQL Server filegroup. Then a database can be built on the filegroup so the data will be spread evenly across all of the drives and RAID controllers. The "drive pool" method depends on RAID to divide data across all physical drives to help ensure parallel access to that data during database server operations.

This drive pool method simplifies SQL Server I/O performance tuning because database administrators know there is only one physical location in which to create database objects. The single pool of drives can be watched for disk queuing and, if necessary, more hard drives can be added to the pool to prevent disk queuing. This method helps optimize for the common case, in which it is unknown what parts of databases may get the most usage. It is better not to have a portion of the total available I/O capacity segregated on another disk partition just because five percent of the time SQL Server might be doing I/O to it. The "single pool of drives" method helps make all available I/O capacity "always" available for SQL Server operations. It also allows I/O operations to be spread across the maximum number of

disks available.

SQL Server log files should *always* be physically separated onto different hard drives from all other SQL Server database files. For SQL Servers managing multiple busy databases that are very busy, the transaction log files for each database should be physically separated from each other to reduce contention.

Because transaction logging is primarily a sequential write I/O, the separation of log files tends to yield a tremendous I/O performance benefit. The disk drives containing the log files can very efficiently perform these sequential write operations if they are not interrupted by other I/O requests. At times, the transaction log will need to be read as part of SQL Server operations, such as replication, rollbacks, and deferred updates. Some implementations use replication as a front end to their data transformation utility as a means of loading new data into the data warehouse in near real time. Administrators of SQL Servers that participate in replication need to make sure that all disks used for transaction log files have sufficient I/O processing power to accommodate the reads that need to occur in addition to the normal log transaction writes.

Additional administration is required to physically segment files and filegroups. The additional effort may prove worthwhile when segmenting for the purposes of isolating and improving access to very active tables or indexes. Some of the benefits are listed below:

- More accurate assessments can be made of the I/O requirements for specific objects, which is not as easy to do when all database objects are placed within one big drive pool.
- Partitioning data and indexes using files and file groups can enhance the administrator's ability to create a more granular backup and restore strategy.
- File and filegroups may be used to maintain the sequential placement of data on disk, thus reducing or eliminating nonsequential I/O activity. This can be extremely important if your available window of time for loading data into the warehouse requires processing be performed in parallel to meet the deadline.
- Physically segmenting files and filegroups may be appropriate during database development and benchmarking so database I/O information can be gathered and applied to capacity planning for the production database server environment.

Objects For Partitioning Consideration

The following areas of SQL Server activity can be separated across different hard drives, RAID controllers, and PCI channels (or combinations of the three):

- Transaction log
- tempdb
- Database
- Tables
- Nonclustered indexes

  Note In SQL Server 2000, Microsoft introduced enhancements to distributed partitioned views that enable the creation of federated databases (commonly referred to as scale-out), which spread resource load and I/O activity across multiple servers. Federated databases are appropriate for some high-end online analytical processing (OLTP) applications, but this approach is not recommended for addressing the needs of a data warehouse.

The physical segregation of SQL Server I/O activity is quite easy to achieve using hardware RAID controllers, RAID hot plug drives, and online RAID expansion. The approach that provides the most flexibility is arranging RAID controllers so that separate RAID channels are associated with the different areas of activity mentioned above. Also, each RAID channel should be attached to a separate RAID hot plug cabinet to take full advantage of online RAID expansion (if available through the RAID controller). Windows logical drive letters are then associated to each RAID array and SQL Server files may be separated between distinct RAID arrays based on known I/O usage patterns.

With this configuration it is possible to relate disk queuing associated with each activity back to a distinct RAID channel and its drive cabinet. If a RAID controller and its drive array cabinet both support online RAID expansion and slots for hot plug hard drives are available in the cabinet, disk queuing issues on that RAID array can be resolved by simply adding more drives to the RAID array until System Monitor reports that disk queuing for that RAID array has reached an acceptable level (ideally less than two for SQL Server files). This can be done while SQL Server is online.

Segregating the Transaction Log

Transaction log files should be maintained on a storage device physically separate from devices that contain data files. Depending on your database recovery model setting, most update activity generates both data device activity and log activity. If both are set up to share the same device, the operations to be performed will compete for the same limited resources. Most installations benefit from separating these competing I/O activities.

Segregating tempdb

SQL Server creates a database, **tempdb**, on every server instance to be used by the server as a shared working area for various activities, including temporary tables, sorting, processing subqueries, building aggregates to support GROUP

BY or ORDER BY clauses, queries using DISTINCT (temporary worktables have to be created to remove duplicate rows), cursors, and hash joins. By segmenting **tempdb** onto its own RAID channel, we enable **tempdb** I/O operations to occur in parallel with the I/O operations of their related transactions. Because **tempdb** is essentially a scratch area and very update intensive, RAID 5 is not as good a choice for **tempdb** – RAID 1 or 0+1 offer better performance. Raid 0, even though it does not provide fault tolerance, can be considered for **tempdb** because **tempdb** is rebuilt every time the database server is restarted. RAID 0 provides the best RAID performance for **tempdb** with the least number of physical drives, but the main concern about using RAID 0 for **tempdb** in a production environment is that SQL Server availability might be compromised if any physical drive failure were to occur, including the drive used for **tempdb**. This can be avoided if **tempdb** is placed on a RAID configuration that provides fault tolerance.

To move the **tempdb** database, use the ALTER DATABASE command to change the physical file location of the SQL Server logical file name associated with **tempdb**. For example, to move **tempdb** and its associated log to the new file locations E:\mssql7 and C:\temp, use the following commands:

```
alter database tempdb modify file (name='tempdev',filename= 'e:\mssql7\tempnew_location.mDF')
alter database tempdb modify file (name='templog',filename= 'c:\temp\tempnew_loglocation.mDF')
```

The master database, **msdb**, and model databases are not used much during production compared to user databases, so it is typically not necessary to consider them in I/O performance tuning considerations. The master database is usually used only for adding new logins, databases, devices, and other system objects.

Database Partitioning

Databases can be partitioned using files and/or filegroups. A filegroup is simply a named collection of individual files grouped together for administration purposes. A file cannot be a member of more than one filegroup. Tables, indexes, text, ntext, and image data can all be associated with a specific filegroup. This means that all their pages are allocated from the files in that filegroup. The three types of filegroups are described below.

Primary filegroup

This filegroup contains the primary data file and any other files not placed into another filegroup. All pages for the system tables are allocated from the primary filegroup.

User-defined filegroup

This filegroup is any filegroup specified using the FILEGROUP keyword in a CREATE DATABASE or ALTER DATABASE statement, or on the Properties dialog box within SQL Server Enterprise Manager.

Default filegroup

The default filegroup contains the pages for all tables and indexes that do not have a filegroup specified when they are created. In each database, only one filegroup at a time can be the default filegroup. If no default filegroup is specified, the default is the primary filegroup.

Files and filegroups are useful for controlling the placement of data and indexes and to eliminate device contention. Quite a few installations also leverage files and filegroups as a mechanism that is more granular than a database in order to exercise more control over their database backup/recovery strategy.

Horizontal Partitioning (Table)

Horizontal partitioning segments a table into multiple tables, each containing the same number of columns but fewer rows. Determining how to partition the tables horizontally depends on how data is analyzed. A general rule of thumb is to partition tables so queries reference as few tables as possible. Otherwise, excessive UNION queries, used to merge the tables logically at query time, can impair performance.

For example, assume business requirements dictate that we store a rolling ten years worth of transactional data in the central fact table of our data warehouse. Ten years of transactional data for our company represents more than one billion rows. A billion of anything is a challenge to manage. Now consider that every year we have to drop the tenth year of data and load the latest year.

A common approach administrators take is to create ten separate, but identically structured tables, each holding one year's worth of data. Then the administrator defines a single union view over top of the ten tables to provide end users with the appearance that all of the data is being housed in a single table. In fact, it is not. Any query posed against the view is optimized to search only the specified years (and corresponding tables). However, the administrator does gain manageability. The administrator can now granularly manage each year of data independently. Each year of data can be loaded, indexed, or maintained on its own. To add a new year is as simple as dropping the view, dropping the table with the tenth year of data, loading and indexing the new year of data, and then redefining the new view to include the new year of data.

When you partition data across multiple tables or multiple servers, queries accessing only a fraction of the data can run

faster because there is less data to scan. If the tables are located on different servers, or on a computer with multiple processors, each table involved in the query can also be scanned in parallel, thereby improving query performance. Additionally, maintenance tasks, such as rebuilding indexes or backing up a table, can execute more quickly.

By using a partitioned view, the data still appears as a single table and can be queried as such without having to reference the correct underlying table manually. Partitioned views are updatable if either of the following conditions is met. For details about partitioned views and their restrictions, see SQL Server Books Online.

- An INSTEAD OF trigger is defined on the view with logic to support INSERT, UPDATE, and DELETE statements.
- The view and the INSERT, UPDATE, and DELETE statements follow the rules defined for updatable partitioned views.

Segregating Nonclustered Indexes

Indexes reside in B-tree structures, which can be separated from their related database tables (except for clustered indexes) by using the ALTER DATABASE command to set up a distinct filegroup. In the example below, the first ALTER DATABASE creates a filegroup. The second ALTER DATABASE adds a file to the newly created filegroup.

```
alter database testdb add filegroup testgroup1
alter database testdb add file (name = 'testfile',
filename = 'e:\mssql7\test1.ndf') to filegroup testgroup1
```

After a filegroup and its associated files have been created, the filegroup can be used to store indexes by specifying the filegroup when the indexes are created.

```
create table test1(col1 char(8))
create index index1 on test1(col1) on testgroup1
```

SP_HELPFILE reports information back about files and filegroups in a given database. SP_HELP <tablename> has a section in its output, which provides information on a table's indexes and their filegroup relationships.

```
sp_helpfile
sp_help test1
```

Parallel Data Retrieval

SQL Server can perform parallel scans of data when running on a computer that has multiple processors. Multiple parallel scans can be executed for a single table if the table is in a filegroup that contains multiple files. Whenever a table is accessed sequentially, a separate thread is created to read each file in parallel. For example, a full scan of a table created on a filegroup that consists of four files will use four separate threads to read the data in parallel. Therefore, creating more files for each filegroup can help increase performance because a separate thread is used to scan each file in parallel. Similarly, when a query joins tables on different filegroups, each table can be read in parallel, thereby improving query performance.

Additionally, any **text**, **ntext**, or **image** columns within a table can be created on a filegroup other than the one that contains the base table.

Eventually, a saturation point is reached when there are too many files and therefore too many parallel threads causing bottlenecks in the disk I/O subsystem. These bottlenecks can be identified by using Windows System Monitor (Performance Monitor in Windows NT 4.0) to monitor the **PhysicalDisk** object and **Disk Queue Length** counter. If the **Disk Queue Length** counter is greater than three, consider reducing the number of files.

It is advantageous to get as much data spread across as many physical drives as possible in order to improve throughput through parallel data access using multiple files. To spread data evenly across all disks, first set up hardware-based disk striping, and then use filegroups to spread data across multiple hardware stripe sets if needed.

Parallel Query Recommendations

SQL Server can automatically execute queries in parallel. This optimizes the query execution in multiprocessor computers. Rather than using one OS thread to execute one query, work is broken down into multiple threads (subject to the availability of threads and memory), and complex queries are completed faster and more efficiently.

The optimizer in SQL Server generates the plan for the query and determines when a query will be executed in parallel. The determination is made based on the following criteria:

- Does the computer have multiple processors?
- Is there enough memory available to execute the query in parallel?
- What is the CPU load on the server?
- What type of query is being run?

When allowing SQL Server to run parallel operations like DBCC and index creation in parallel, the server resources become stressed, and you might see warning messages when heavy parallel operations are occurring. If warning messages about insufficient resources appear frequently in the server error log, consider using System Monitor (Performance Monitor in Windows NT 4.0) to investigate what resources are available, such as memory, CPU usage, and I/O usage.

Do not run heavy queries that are executed in parallel when there are active users on the server. Try executing maintenance jobs such as DBCC and INDEX creation during offload times. These jobs can be executed in parallel. Monitor the disk I/O performance. Observe the disk queue length in System Monitor (Performance Monitor in Windows NT 4.0) to make decisions about upgrading your hard disks or redistributing your databases onto different disks. Upgrade or add more processors if the CPU usage is very high.

The following server configuration options can affect parallel execution of the queries:

- cost threshold for parallelism
- max degree of parallelism
- max worker threads
- query governor cost limit

Optimizing Data Loads

There are multiple tips and techniques to keep in mind for accelerating your data loading activities. The techniques will likely vary based on whether you are doing initial data loads or incremental data loads. Incremental loads in general are more involved and restrictive. The techniques you choose might also be based on factors outside your control. Processing window requirements, your chosen storage configuration, limitations of your server hardware, and so on, can all impact the options available to you.

There are a number of common things to keep in mind when performing both initial data loads and incremental data loads. The following subjects will be discussed in detail below:

- Choosing an appropriate database recovery model
- Using **bcp**, BULK INSERT, or the bulk copy API
- Controlling the Locking behavior
- Loading data in parallel
- Miscellaneous, including:
  - Bypassing referential integrity checks (constraints & triggers)
  - Loading presorted data
  - Effects of removing indexes

Choosing an Appropriate Database Recovery Model

We discussed database recovery models in the section "Configuration Options That Impact Performance." It is important to remember that the recovery model you choose can have a significant impact on the amount of time needed to perform your data load. Basically, these recovery models control the amount of data that will be written out to the transaction log. This is important because performing write operations to the transaction log essentially doubles the workload.

Logged and Minimally Logged Bulk Copy Operations

When using the full recovery model, all row-insert operations performed by one of the bulk data load mechanisms (discussed below) are logged in the transaction log. For large data loads, this can cause the transaction log to fill rapidly. To help prevent the transaction log from running out of space you can perform minimally logged bulk copy operation. Whether a bulk copy is performed as logged or nonlogged is not specified as part of the bulk copy operation; it is dependent on the state of the database and the table involved in the bulk copy. A nonlogged bulk copy occurs if all the following conditions are met:

- The recovery model is Simple or Bulk-Logged or the database option **select into/bulkcopy** is set to true.
- The target table is not being replicated.
- The target table has no indexes, or if the table has indexes, it is empty when the bulk copy starts.
- The TABLOCK hint is specified using **bcp_control** with *eOption* set to BCPHINTS.

Any bulk copy into an instance of SQL Server that does not meet these conditions is fully logged.

On initial data loads you should always operate under the Bulk-Logged or Simple recovery model. For incremental data loads, consider using bulk-logged as long as the potential for data loss is low. Because many data warehouses are primarily read-only or have a minimal amount of transaction activity, setting the database recovery model to bulk-logged may pose no problem.

Using bcp, BULK INSERT, or the Bulk Copy APIs

Two mechanisms exist inside SQL Server to address the needs of bulk movement of data. The first mechanism is the **bcp** utility. The second is the BULK INSERT statement. **bcp** is a command prompt utility that copies data both into or

out of SQL Server. With SQL Server 2000, the **bcp** utility was rewritten using the ODBC bulk copy application programming interface (API). Earlier versions of the **bcp** utility were written using the DB-Library bulk copy API.

BULK INSERT is a Transact-SQL statement included with SQL Server that can be executed from within the database environment. Unlike **bcp**, BULK INSERT can only pull data into SQL Server. It cannot push data out. An advantage to using BULK INSERT is that it can copy data into an instance of SQL Server using a Transact-SQL statement, rather than having to shell out to the command prompt.

A third option, which often appeals to programmers, is the bulk copy APIs. These APIs enable programmers to move data into or out of SQL Server using ODBC, OLE DB, SQL-DMO, or even DB-Library-based applications.

All of these options enable you to exercise control over the batch size. Unless you are working with small volumes of data, it is good to get in the habit of specifying a batch size for recoverability reasons. If none is specified, SQL Server commits all rows to be loaded as a single batch. For example, you attempt to load 1,000,000 rows of new data into a table. The server suddenly loses power just as it finishes processing row number 999,999. When the server recovers, those 999,999 rows will need to be rolled back out of the database before you attempt to reload the data. By specifying a batch size of 10,000 you could have saved yourself significant recovery time because you would have only had to rollback 9,999 rows instead of 999,999. This is because you would have already committed rows 1-990,000 to the database. Also, without a specified batch size, you would have to restart the load processing back at row 1 in order to reload the data. With the specified batch size of 10,000 rows, you could simply restart the load processing at row 990,001, effectively bypassing the 990,000 rows already committed.

Controlling the Locking Behavior

The **bcp** utility and BULK INSERT statement accept the **TABLOCK** hint, which allows the user to specify the locking behavior to be used. **TABLOCK** specifies that a bulk update table-level lock will be taken for the duration of the bulk copy operation. Using **TABLOCK** can improve performance of the bulk copy operation due to reduced lock contention on the table. This setting has significant implications when parallel loads are being processed against a single table (discussed in next section).

For example, to bulk copy data from the Authors.txt data file to the **authors2** table in the **pubs** database, specifying a table-level lock, execute from the command prompt:

```
bcp pubs..authors2 in authors.txt -c -t, -Sservername -Usa -Ppassword -h "TABLOCK"
```

Alternatively, you could use the BULK INSERT statement from a query tool, such as SQL Query Analyzer, to bulk copy data, as in this example:

```
BULK INSERT pubs..authors2 FROM 'c:\authors.txt'
WITH (
DATAFILETYPE = 'char',
FIELDTERMINATOR = ',',
TABLOCK
)
```

If **TABLOCK** is not specified, the default locking uses row-level locks, unless the **table lock on bulk load** option is set to **on** for the table. Using the **table lock on bulk load** option with the **sp_tableoption** command is an alternative way to set the locking behavior for a table during a bulk load operation.

Table lock on bulk load	Table locking behavior
Off	Row-level locks used
On	Table-level lock used

Note If the **TABLOCK** hint is specified, it overrides the setting declared using the **sp_tableoption** for the duration of the bulk load.

Loading Data in Parallel

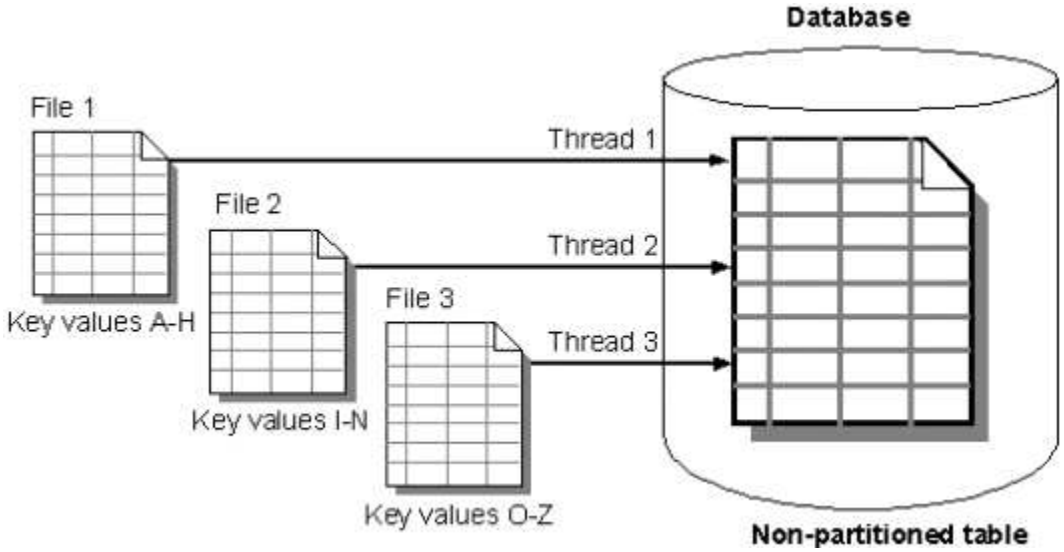
Parallel Load - Nonpartitioned Table

It is possible to perform parallel data loads into a single, nonpartitioned table using any of the bulk data load mechanisms in SQL Server. This is done by using running multiple data loads simultaneously. The data to be loaded in parallel needs to be split into separate files (data sources for the bulk insert API) prior to beginning the load. Then all the separate load operations can be initiated at the same time so that the data loads in parallel.

For example, assume you need to load a consolidation database for a service company that operates in four global regions, each reporting report hours billed to clients on a monthly basis. For a large service organization, this could represent a large amount of transactional data that needs to be consolidated. If each of the four reporting regions provided a separate file, it would be possible using the methodology described earlier to load all four files simultaneously into a single table.

Note The number of parallel threads (loads) you process in parallel should not exceed the number of processors available to SQL Server.

The following illustration shows parallel loading on a nonpartitioned table.



If your browser does not support inline frames, click here to view on a separate page.

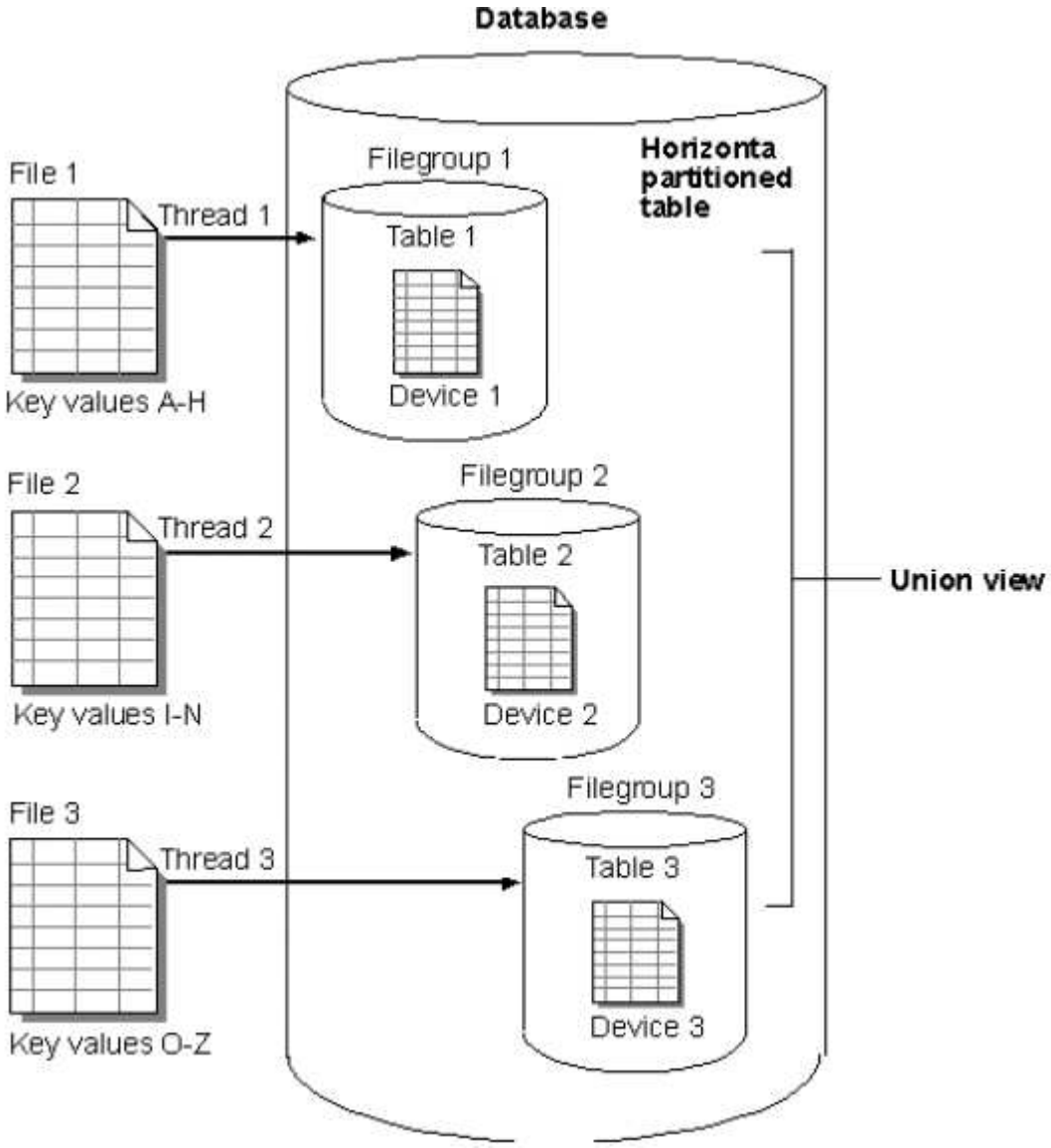
Parallel Load - Horizontal Partitioning (Table)

This section focuses on how horizontal partitioned tables can be used to improve the speed of your data loads. In a previous section, we discussed loading data from multiple files into a single (nonpartitioned) table. Horizontal partitioning of the table offers an opportunity to possibly improve the contiguousness of your data as well as speeding up the load process by reducing device contention. Though the above figure shows the data being loaded into different sections of the table, this may not be an accurate depiction. If all three threads in the above load were being processed simultaneously, the extents taken for the table would likely end up intermingled. The intermingling of the data could result in less than optimal performance when the data is retrieved. This is because the data was not stored in physically contiguous order, which could cause the system to access it using nonsequential I/O.

Building a clustered index over this table would solve the problem, because the data would be read in, sorted into the key order, and written back out in contiguous order. However, the reading, sorting, deletion of the old data, and writing back out of the newly sorted data can be a time consuming task (see loading presorted data below). To avoid this intermingling, consider using filegroups to reserve chunks of contiguous space where you can store large tables. Many installations also use filegroups to segregate index data away from table data.

To illustrate, assume a data warehouse that is allocated onto one large physical partition. Any load operations performed in parallel to that database are likely to cause the affected data/index pages to be stored in a noncontiguous (intermingled) state. What sort of operations? Any operation that modifies the data will cause the data to become noncontiguous. Initial data loads, incremental data loads, index creation, index maintenance, inserts, updates, deletes, and so on are all activities that one might be tempted to perform in parallel in order to meet processing window requirements.

The following illustration shows partitioning a table across multiple filegroups.



If your browser does not support inline frames, click here to view on a separate page.

Loading Pre-Sorted Data

Earlier versions of SQL Server included an option that allowed you to specify a SORTED_DATA option when creating an index. This has been eliminated in SQL Server 2000. The reason for specifying this option as part of your CREATE INDEX statement in earlier versions is that it enabled you to avoid a sort step in the index creation process. By default in SQL Server, when creating a clustered index, the data in the table is sorted during the processing. To get the same effect with SQL Server 2000, consider creating the clustered index before bulk loading the data. Bulk operations in SQL Server 2000 use enhanced index maintenance strategies to improve the performance of data importation on tables having a preexisting clustered index, and to eliminate the need for resorting data after the import.

Impact of FILLFACTOR and PAD_INDEX on Data Loads

FILLFACTOR and PAD_INDEX are explained more fully under the section titled "Indexes and Index Maintenance." The key thing to remember about both FILLFACTOR and PAD_INDEX is that leaving them set to default, when creating an index, may cause SQL Server to perform more writes and read I/O operations than are needed to store the data. This is especially true of data warehouses having very little write activity going on in them, but high amounts of read activity. To get SQL Server to pack more data into a single page of the data or index pages, you can specify a particular FILLFACTOR when creating the index. It is a good idea to specify the PAD_INDEX when providing an overriding FILLFACTOR value.

General Guidelines for Initial Data Loads

While Loading Data

- Remove indexes (one exception might be in loading pre-sorted data – see above)
- Use BULK INSERT, **bcp** or bulk copy API
- Parallel load using partitioned data files into partitioned tables
- Run one load stream for each available CPU
- Set Bulk-Logged or Simple Recovery model
- Use the TABLOCK option

After Loading Data

- Create indexes
- Switch to the appropriate recovery model
- Perform backups

General Guidelines for Incremental Data Loads

- Load data with indexes in place.
- Performance and concurrency requirements should determine locking granularity (**sp_indexoption)**.
- Change from Full to Bulk-Logged Recovery model unless there is an overriding need to preserve point-in-time recovery, such as online users modifying the database during bulk loads. Read operations should not affect bulk loads.

Indexes and Index Maintenance

I/O characteristics of the hardware devices on the server have been discussed. Now the discussion will move to how SQL Server data and index structures are physically placed on disk drives. Index placement is likely to be the single biggest influence you can have over your data warehouse to improve performance once your design is set.

Types of Indexes in SQL Server

Although SQL Server 2000 introduced several new types of indexes, all of them are based on two core forms. The two core forms are a clustered index or a nonclustered index format. The two primary types of indexes available to database designers in SQL Server are:

- Clustered indexes.
- Nonclustered indexes.

Additional variations of the two primary types include:

- Unique indexes.
- Indexes on computed columns.
- Indexed views.
- Full text indexes.

Each index type mentioned above will be described in detail in the following sections below except for Full text indexes. Full text indexing is a special case unlike other database indexes and is not covered in this document. An indexed view is a new type of index introduced in SQL Server 2000 that should prove to be of particular interest to the data warehousing audience. Another new feature introduced in SQL Server 2000 is the ability to create indexes in either ascending or descending order.

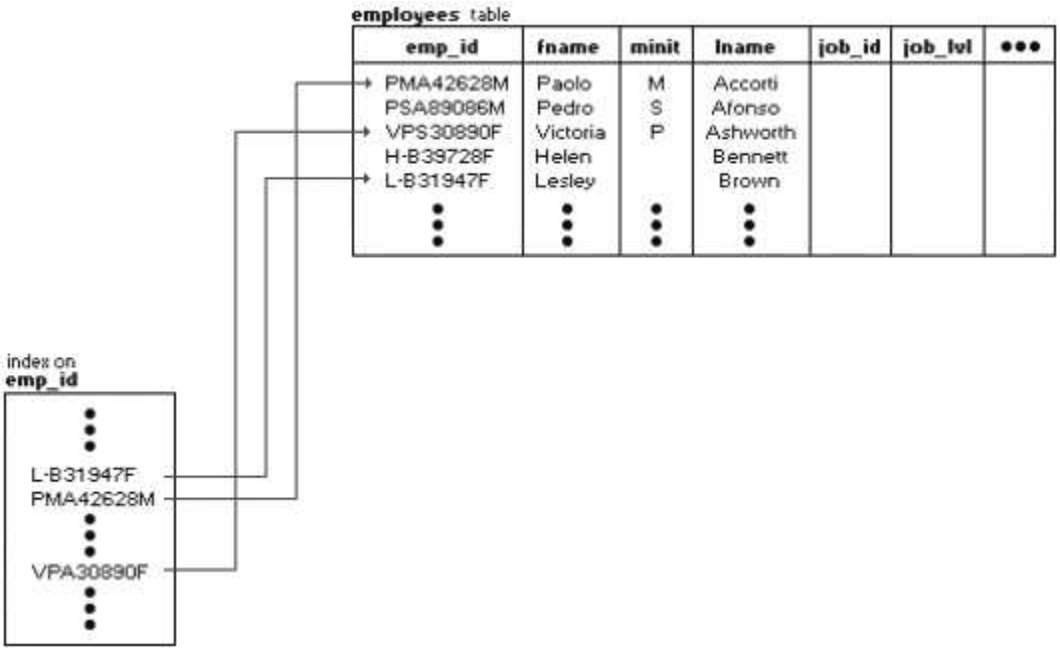
How Indexes Work

Indexes in databases are similar to indexes in books. In a book, an index allows you to find information quickly without reading the entire book. In a database, an index allows the database program to find data in a table without scanning the entire table. An index in a book is a list of words with the page numbers that contain each word. An index in a database is a list of values in a table with the storage locations of rows in the table that contain each value.

Indexes can be created on either a single column or a combination of columns in a table and are implemented in the form of B-trees. An index contains an entry with one or more columns (the search key) from each row in a table. A B-tree is stored in sorted order on the search key in either ascending or descending order (depending on the option chosen when the index is created), and can be searched efficiently on any leading subset of that search key. For example, an index on columns **A**, **B**, **C** can be searched efficiently on **A**, on **A**, **B**, and **A**, **B**, **C**.

When you create a database and tune it for performance, you should create indexes for the columns used in queries to find data. In the **pubs** sample database provided with SQL Server, the **employee** table has an index on the **emp_id** column. When someone executes a statement to find data in the **employee** table based on a specified **emp_id** value,

SQL Server query processor recognizes the index for the emp_id column and uses the index to find the data. The following illustration shows how the index stores each **emp_id** value and points to the rows of data in the table with the corresponding value.



If your browser does not support inline frames, click here to view on a separate page.

The performance benefits of indexes, however, don't come without a cost. Tables with indexes require more storage space in the database. Also, commands that insert, update, or delete data can take longer and require more processing time to maintain the indexes. When you design and create indexes, you should ensure that the performance benefits outweigh the extra cost in storage space and processing resources.

Index Intersection

A unique feature found inside the SQL Server query processor is the ability to perform index intersection. This is a special form of index covering, which we explain in detail later, but index intersection bears mentioning now for two reasons. First, it is a technique that may influence your index design strategy. Second, this technique can possibly reduce the number of indexes you need, which can save significant disk space for very large databases.

Index intersection allows the query processor to use multiple indexes to solve a query. Most database query processors use only one index when attempting to resolve a query. SQL Server can combine multiple indexes from a given table or view, build a hash table based on those multiple indexes, and utilize the hash table to reduce I/O for a given query. The hash table that results from the index intersection becomes, in essence, a covering index and provides the same I/O performance benefits that covering indexes do. Index intersection provides greater flexibility for database user environments in which it is difficult to predetermine all of the queries that will be run against the database. A good strategy in this case is to define single-column, nonclustered indexes on all the columns that will be frequently queried and let index intersection handle situations where a covered index is needed.

The following example makes use of index intersection:

```
Create index Indexname1 on Table1(col2)
Create index Indexname2 on Table1(col3)
Select col3 from table1 where col2 = 'value'
```

When the previous query is performed, the indexes can be combined to quickly and efficiently resolve the query.

Index Architecture In SQL Server

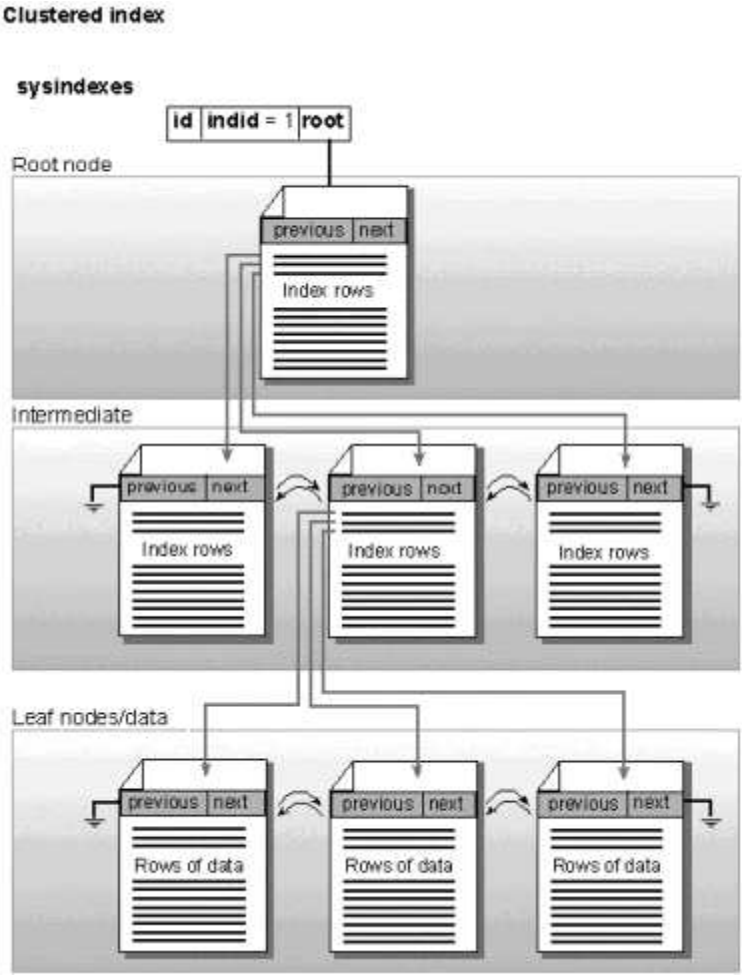
All indexes in SQL Server are physically built upon a B-tree index structures, which are stored on 8-KB index pages. Each index page has a page header followed by the index rows. Each index row contains a key value and a pointer to either a lower-level index page or an actual data row. Each page in an index is also referred to as an index node. The top node of the B-tree is called the root node. The bottom layer of nodes in an index are called the leaf nodes. Any index levels between the root and the leaves are collectively known as intermediate levels or nodes. Pages in each level of the index are linked together in a doubly-linked list.

SQL Server data and index pages are both 8 KB in size. SQL Server data pages contain all of the data associated with a row of a table, with the possible exception of text and image data. In the case of text and image data the SQL Server

data page that contains the row associated with the text or image column will contain, by default, a pointer to a binary tree (or B-tree) structure of one or more 8-KB pages that contain the text or image data. A new feature in SQL Server 2000 is the ability to store small text and image values in-row, which means that small text or image columns will be stored on the data page. This feature can reduce I/O because the additional I/O required to fetch corresponding image or text data can be avoided. For information about how to set a table to store text or images in row, see SQL Server Books Online.

Clustered Indexes

Clustered indexes are very useful for retrieving ranges of data values from a table. Nonclustered indexes are ideally suited for targeting specific rows for retrieval, whereas clustered indexes are ideally suited for retrieving a range of rows. However, adhering to this simple logic for determining which type of index to create is not always successful. This is because only one clustered index is allowed for each table. There is a simple physical reason for this. While the upper parts (nonleaf levels) of the clustered index B-tree structure are organized just like their nonclustered counterparts, the bottom level of a clustered index is made of the actual 8-KB data pages from the table. An exception to this is when a clustered index is created over the top of a view. Because indexed views will be explained below, we will discuss clustered indexes being created on actual tables. When a clustered index is created on a table, the data associated with that table is read, sorted, and physically stored back to the database in the same order as the index search key. Because data for the table can only be persisted to storage in one order without causing duplication, the restriction of one clustered index applies. The following diagram depicts the storage for a clustered index.



Clustered Indexes and Performance

There are some inherent characteristics of clustered indexes that affect performance.

Retrieval of SQL Server data based on key search with a clustered index requires no pointer jump (involving a likely nonsequential change of location on the hard disk) in order to retrieve the associated data page. This is because the leaf level of the clustered index is, in fact, the associated data page.

As mentioned previously, the leaf level (and consequentially the data for the table or indexed view) is physically sorted and stored in the same order as the search key. Because the leaf level of the clustered index contains the actual 8-KB data pages of the table, the row data of the entire table is physically arranged on the disk drive in the order determined by the clustered index. This provides a potential I/O performance advantage when fetching a significant number of rows

from this table (at least greater than 64 KB) based on the value of the clustered index, because sequential disk I/O is being used (unless page splitting is occurring on this table, which will be discussed elsewhere in the section titled "FILLFACTOR and PAD_INDEX"). That is why it is important to pick the clustered index on a table based on a column that will be used to perform range scans to retrieve a large number of rows.

The fact that the rows for table associate with a clustered index have to be sorted and stored in the same order as the index search key has the following implications:

- When you create a clustered index, the table is copied, the data in the table is sorted, and then the original table is deleted. Therefore, enough empty space must exist in the database to hold a copy of the data.
- By default, the data in the table is sorted when the index is created. However, if the data is already sorted in the correct order, the sort operation is automatically skipped. This can have a dramatic effect in speeding up the index creation process.
- Whenever possible, you should load your data into the table in the same order as the search key you intend to use to build the clustered index. On large tables, like those that often characterize data warehouses, this approach will dramatically speed up your index creation, allowing you to reduce the amount of time needed to process initial data load(s). This same approach can be taken when dropping and rebuilding a clustered index, as long as the rows of the table remain in sorted order during the time that the clustered index is not in place. If any rows are not correctly sorted, the operation cancels, an appropriate error message will be given, and the index will not be created.
- Also, building clustered indexes on sorted data requires much less I/O. This is because the data does not have to be copied, sorted, stored back to the database, then the old table data deleted. Instead, the data is left in the extents where it was originally allocated. Index extents are simply added to the database to store top and intermediate nodes.

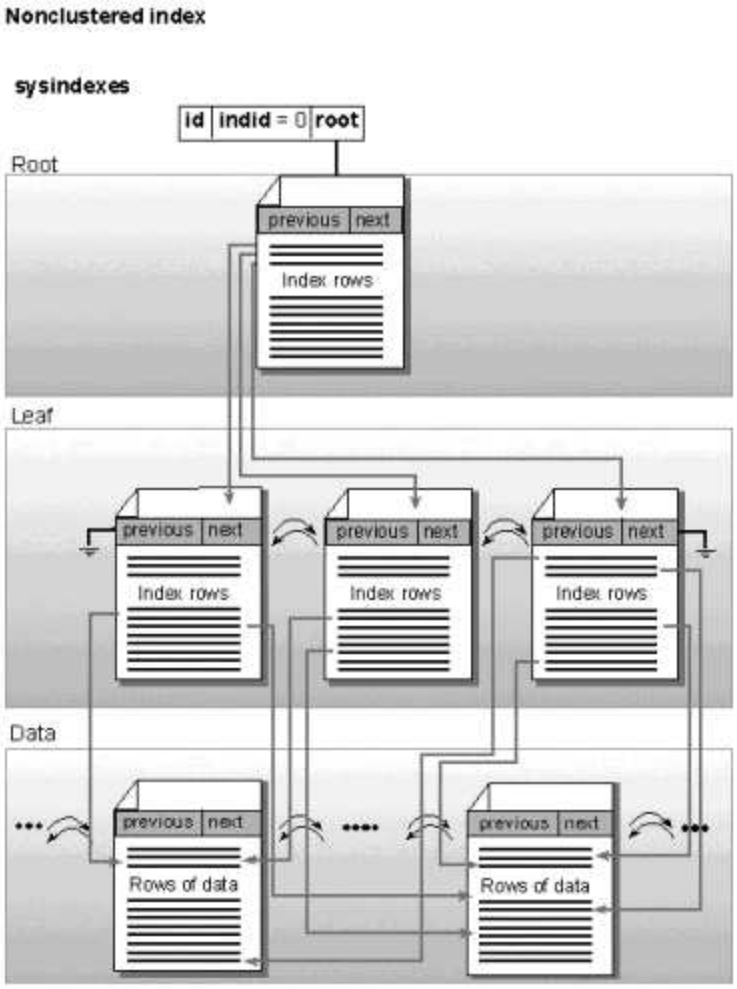
  Note The preferred way to build indexes on large tables is to start with the clustered index and then build the nonclustered indexes. In this way, no nonclustered indexes will need to be rebuilt due to the data moving. When dropping all indexes, drop the nonclustered indexes first and the clustered index last. That way, no indexes need to be rebuilt.

Nonclustered Indexes

Nonclustered indexes are most useful for fetching few rows with good selectivity from large SQL Server tables based on a specific key value. As mentioned previously, nonclustered indexes are binary trees formed out of 8-KB index pages. The bottom, or leaf level, of the binary tree of index pages contains all the data from the columns that comprised that index. When a nonclustered index is used to retrieve information from a table based on a match with the key value, the index B-tree is traversed until a key match is found at the leaf level of the index. A pointer jump is made if columns from the table are needed that did not form part of the index. This pointer jump will likely require a nonsequential I/O operation on the disk. It might even require the data to be read from another disk, especially if the table and its accompanying index B-trees are large in size. If multiple pointers lead to the same 8-KB data page, less of an I/O performance penalty will be paid because it is only necessary to read the page into data cache once. For each row returned for an SQL query that involves searching with a nonclustered index, at least one pointer jump is required.

Note The overhead associated with each pointer jump is the reason that nonclustered indexes are better suited for processing queries that return only one or a few rows from a table. Queries that require a range of rows are better served with a clustered index.

The following diagram shows the storage for a nonclustered index. Notice the added leaf level that points to the corresponding data pages. That is where the added pointer jump takes place when using a nonclustered index to access table data as opposed to using a clustered index. For more information on nonclustered indexes, see SQL Server Books Online.



Unique Indexes

Both clustered and nonclustered indexes can be used to enforce uniqueness within a table by specifying the UNIQUE keyword when creating an index on an existing table. Using a UNIQUE constraint is another way to ensure uniqueness within a table. UNIQUE constraints, like unique indexes, enforce the uniqueness of the values in a set of columns. In fact, the assignment of a UNIQUE constraint automatically creates an underlying unique index to facilitate the enforcement of the constraint. Because the uniqueness can be defined and documented as part of the CREATE TABLE statement, a UNIQUE constraint is often preferred over the creation of a separate unique index.

Indexes on Computed Columns

SQL Server 2000 introduced the capability to create indexes on computed columns. This is a handy feature for situations where queries are commonly submitted and computed columns are routinely provided, but the administrator would prefer not to persist the data into an actual column of a table simply to allow the creation of an index. In this case, computed columns can be referenced to create an index as long as the computed column satisfies all conditions required for indexing. Among other restrictions, the computed column expression must be deterministic, precise, and must not evaluate to **text**, **ntext**, or **image** data types.

Deterministic

A nondeterministic user-defined function cannot be invoked by either a view or computed column if you want to create an index on the view or computed column. All functions are deterministic or nondeterministic:

- Deterministic functions always return the same result any time they are called with a specific set of input values.
- Nondeterministic functions may return different results each time they are called with a specific set of input values.

For example, the DATEADD built-in function is deterministic because it always returns a predictable result for a given set of argument values passed in via its three input parameters. GETDATE is not deterministic. While it is always invoked with the same argument value, the value it returns changes each time executed.

Precise

A computed column expression is precise if:

- It is not an expression of the **float** data type.
- It does not use in its definition a float data type. For example, in the following statement, column **y** is **int** and deterministic, but not precise.

```
CREATE TABLE t2 (a int, b int, c int, x float,
y AS CASE x
WHEN 0 THEN a
WHEN 1 THEN b
ELSE c
END)
```

The **IsPrecise** property of the COLUMNPROPERTY function reports whether a **computed_column_expression** is precise.

Note Any **float** expression is considered nonprecise and cannot be a key of an index; a **float** expression can be used in an indexed view but not as a key. This is true also for computed columns. Any function, expression, user-defined function, or view definition is considered nondeterministic if it contains any **float** expressions, including logical ones (comparisons).

Creation of an index on a computed column or view may cause the failure of an INSERT or UPDATE operation that previously operated correctly. Such a failure may take place when the computed column results in an arithmetic error. For example, although computed column c in the following table will result in an arithmetic error, the INSERT statement will work:

```
CREATE TABLE t1 (a int, b int, c AS a/b)
GO
INSERT INTO t1 VALUES ('1', '0')
GO
```

If, instead, after creating the table, you create an index on computed column **c**, the same INSERT statement now will fail.

```
CREATE TABLE t1 (a int, b int, c AS a/b)
GO
CREATE UNIQUE CLUSTERED INDEX Idx1 ON t1.c
GO
INSERT INTO t1 VALUES ('1', '0')
GO
```

Indexed Views

Indexed views are views whose results are persisted in the database and indexed for fast access. As with any other views, indexed views depend on base tables for their data. Such dependency means that if you change a base table contributing to an indexed view, the indexed view might become invalid. For example, renaming a column that contributes to a view invalidates the view. To prevent such problems, SQL Server supports creating views with schema binding. Schema binding prohibits any table or column modification that would invalidate the view. Any indexed view you create with the View Designer automatically gets schema binding, because SQL Server requires that indexed views have schema binding. Schema binding does not mean you cannot modify the view; it means you cannot modify the underlying tables or views in ways that would change the view's result set. Also, indexed views, like indexes on computed columns, must be deterministic, precise, and must not contain **text**, **ntext**, or **image** columns.

Indexed views work best when the underlying data is infrequently updated. The maintenance of an indexed view can be higher than the cost of maintaining a table index. If the underlying data is updated frequently, then the cost of maintaining the indexed view data may outweigh the performance benefits of using the indexed view.

Indexed views improve the performance of these types of queries:

- Joins and aggregations that process many rows.
- Join and aggregation operations that are frequently performed by many queries.

  For example, in an OLTP database that is recording inventories, many queries would be expected to join the **Parts**, **PartSupplier**, and **Suppliers** tables. Although each query that performs this join may not process many rows, the overall join processing of hundreds of thousands of such queries can be significant. Because these relationships are not likely to be updated frequently, the overall performance of the entire system could be improved by defining an indexed view that stores the joined results.
- Decision support workloads.
- Analysis systems are characterized by storing summarized, aggregated data that is infrequently updated. Further aggregating the data and joining many rows characterizes many decision support queries.

Indexed views usually do not improve the performance of these types of queries:

- OLTP systems with many writes.

- Databases with many updates.
- Queries that do not involve aggregations or joins.
- Aggregations of data with a high degree of cardinality for the key. A high degree of cardinality means the key contains many different values. A unique key has the highest possible degree of cardinality because every key has a different value. Indexed views improve performance by reducing the number of rows a query has to access. If the view result set has almost as many rows as the base table, then there is little performance benefit from using the view. For example, consider this query on a table that has 1,000 rows:

  ```
  SELECT PriKey, SUM(SalesCol)
  FROM ExampleTable
  GROUP BY PriKey
  ```

  If the cardinality of the table key is 100, an indexed view built using the result of this query would have only 100 rows. Queries using the view would, on average, need one tenth of the reads needed against the base table. If the key is a unique key, the cardinality of the key is 1000 and the view result set returns 1000 rows. A query has no performance gain from using this indexed view instead of directly reading the base table.
- Expanding joins, which are views whose result sets are larger than the original data in the base tables.

Design your indexed views to satisfy multiple operations. Because the optimizer can use an indexed view even when the view itself is not specified in the FROM clause, a well-designed indexed view can speed the processing of many queries. For example, consider creating an index on this view:

```
CREATE VIEW ExampleView (PriKey, SumColx, CountColx)
AS
SELECT PriKey, SUM(Colx), COUNT_BIG(Colx)
FROM MyTable
GROUP BY PriKey
```

Not only does this view satisfy queries that directly reference the view columns, it can also be used to satisfy queries that query the underlying base table and contain expressions such as SUM(Colx), COUNT_BIG(Colx), COUNT(Colx), and AVG(Colx). All such queries will be faster because they only have to retrieve the small number of rows in the view rather than reading the full number of rows from the base tables.

The first index created on a view must be a unique clustered index. After the unique clustered index has been created, you can create additional nonclustered indexes. The naming conventions for indexes on views are the same as for indexes on tables. The only difference is that the table name is replaced with a view name.

All indexes on a view are dropped if the view is dropped. All nonclustered indexes on the view are dropped if the clustered index is dropped. Nonclustered indexes can be dropped individually. Dropping the clustered index on the view removes the stored result set, and the optimizer returns to processing the view like a standard view.

Although only the columns that make up the clustered index key are specified in the CREATE UNIQUE CLUSTERED INDEX statement, the complete result set of the view is stored in the database. As in a clustered index on a base table, the B-tree structure of the clustered index contains only the key columns, but the data rows contain all of the columns in the view result set.

Note You can create indexed views in any edition of SQL Server 2000. In SQL Server 2000 Enterprise Edition, the indexed view will be considered automatically by the query optimizer. To use an indexed view in all other editions, the NOEXPAND hint must be used.

Covering Indexes

A covering index is a nonclustered index that is built upon all of the columns required to satisfy an SQL query, both in the selection criteria and the WHERE predicate. Covering indexes can save a huge amount of I/O, and hence bring a lot of performance to a query. But it is necessary to balance the costs of creating a new index (with its associated B-tree index structure maintenance) against of the I/O performance gain the covering index will bring. If a covering index will greatly benefit a query or set of queries that will be run very often on SQL Server, the creation of that covering index may be worth it.

The following example demonstrates use of a covering index intersection:

```
Create index indexname1 on table1(col2,col1,col3).
Select col3 from table1 where col2 = 'value'
```

When the above query is performed, the values needed from the underlying table could be retrieved quickly by just reading the smaller index pages and the query would be resolve quite efficiently. In general, if the covering index is small, in terms of the number of bytes from all the columns in the index compared to the number of bytes in a single row of that table, and it is certain that the query taking advantage of the covered index will be executed frequently, it may make sense to use a covering index.

Index Selection

The choice of indexes significantly affects the amount of disk I/O generated and, subsequently, performance. Nonclustered indexes are good for retrieval of a small number of rows and clustered indexes are good for range-scans.

The following guidelines can be helpful in choosing what type of index to use:

- Try to keep indexes as compact (fewest number of columns and bytes) as possible. This is especially true for clustered indexes because nonclustered indexes will use the clustered index as its method for locating row data.
- In the case of nonclustered indexes, selectivity is important. If a nonclustered index is created on a large table with only a few unique values, use of that nonclustered index will not save much I/O during data retrieval. In fact, using the index will likely cause much more I/O than simply performing a sequential table scan. Good candidates for a nonclustered index include invoice numbers, unique customer numbers, social security numbers, and telephone numbers.
- Clustered indexes perform better than nonclustered indexes for queries that involve range scans or when a column is frequently used to join with other tables. The reason is because the clustered index physically orders the table data, allowing for sequential 64-KB I/O on the key values. Some possible candidates for a clustered index include states, company branches, date of sale, zip codes, and customer district. Only one clustered index can be created for a table; if a table contains a column from which typical queries frequently fetch large sequential ranges and columns of unique values, use the clustered index on the first column and nonclustered indexes on the columns of unique values. The key question to ask when trying to choose the best column on each table to create the clustered index on is, "Will there be a lot of queries that need to fetch a large number of rows based on the order of this column?" The answer is very specific to each user environment. One company may do a lot of queries based on ranges of dates, whereas another company may do a lot of queries based on ranges of bank branches.

Index Creation and Parallel Operations

The query plans built for the creation of indexes allow parallel, multi-threaded index create operations on computers with multiple microprocessors in SQL Server Enterprise and Developer editions.

SQL Server uses the same algorithms to determine the degree of parallelism (the total number of separate threads to run) for create index operations as it does for other Transact-SQL statements. The only difference is that the CREATE INDEX, CREATE TABLE, or ALTER TABLE statements that create indexes do not support the MAXDOP query hint. The maximum degree of parallelism for an index creation is subject to the **max degree of parallelism** server configuration option, but you cannot set a different MAXDOP value for individual index creation operations.

When SQL Server builds a create index query plan, the number of parallel operations is set to the lowest value of:

- The number of microprocessors, or CPUs in the computer.
- The number specified in the **max degree of parallelism** server configuration option.
- The number of CPUs not already over a threshold of work performed for SQL Server threads.

For example, on a computer with eight CPUs, but where the **max degree of parallelism** option is set to 6, no more than six parallel threads are generated for an index creation. If five of the CPUs in the computer exceed the threshold of SQL Server work when an index creation execution plan is built, the execution plan specifies only three parallel threads.

The main phases of parallel index creation include:

- A coordinating thread quickly and randomly scans the table to estimate the distribution of the index keys. The coordinating thread establishes the key boundaries that will create a number of key ranges equal to the degree of parallel operations, where each key range is estimated to cover similar numbers of rows. For example, if there are four million rows in the table, and the **max degree of parallelism** option is set to 4, the coordinating thread will determine the key values that delimit four sets of rows with one million rows in each set.
- The coordinating thread dispatches a number of threads equal to the degree of parallel operations, and waits for these threads to complete their work. Each thread scans the base table using a filter that retrieves only rows with key values within the range assigned to the thread. Each thread builds an index structure for the rows in its key range.

After all the parallel threads have completed, the coordinating thread connects the index subunits into a single index. Individual CREATE TABLE or ALTER TABLE statements can have multiple constraints that require the creation of an index. These multiple index creation operations are performed in series, although each individual index creation operation may be performed as a parallel operation on a computer with multiple CPUs.

Index Maintenance

When you create an index in the database, the index information used by queries is stored in index pages. The sequential index pages are chained together by pointers from one page to the next. When changes are made to the data that affect the index, the information in the index can become scattered in the database. Rebuilding an index reorganizes the storage of the index data (and table data in the case of a clustered index) to remove fragmentation. This can improve disk performance by reducing the number of page reads required to obtain the requested data.

Fragmentation occurs when large amounts of insert activity or updates, which modify the search key value of the clustered index, are performed. For this reason, it is important to try to maintain open space on index and data pages to prevent pages from splitting. Page splitting occurs when an index page or data page can no longer hold any new

rows and a row needs to be inserted into the page because of the logical ordering of data defined in that page. When this occurs, SQL Server needs to divide up the data on the full page and move approximately half of the data to a new page so that both pages will have some open space. Because this consumes system resources and time, doing it frequently is not recommended.

When indexes are initially built, SQL Server attempts to place the index B-tree structures on pages that are physically contiguous; this allows for optimal I/O performance when scanning the index pages using sequential I/O. When page splitting occurs and new pages need to be inserted into the logical B-tree structure of the index, SQL Server must allocate new 8-KB index pages. If this occurs somewhere else on the hard drive, it breaks up the physically sequential nature of the index pages. This can cause I/O operations to switch from being performed sequentially to nonsequentially. It can also dramatically reduce performance. Excessive amounts of page splitting should be resolved by rebuilding your index or indexes to restore the physically sequential order of the index pages. This same behavior can be encountered on the leaf level of the clustered index, thereby affecting the data pages of the table.

In System Monitor, pay particular attention to "SQL Server: Access Methods – **Page Splits/sec**." Nonzero values for this counter indicate that page splitting is occurring and that further analysis should be done with DBCC SHOWCONTIG.

The **DBCC SHOWCONTIG** command can also be used to reveal whether excessive page splitting has occurred on a table. Scan Density is the key indicator that **DBCC SHOWCONTIG** provides. It is good for this value to be as close to 100 percent as possible. If this value is significantly below 100 percent, consider running maintenance on the problem indexes.

DBCC INDEXDEFRAG

One index maintenance option is to use a new statement (DBCC INDEXDEFRAG), which was introduced in SQL Server 2000. DBCC INDEXDEFRAG can defragment clustered and nonclustered indexes on tables and views. DBCC INDEXDEFRAG defragments the leaf level of an index so the physical order of the pages matches the left-to-right logical order of the leaf nodes, thus improving index-scanning performance.

DBCC INDEXDEFRAG also compacts the pages of an index, taking into account the FILLFACTOR specified when the index was created. Any empty pages created as a result of this compaction will be removed.

If an index spans more than one file, DBCC INDEXDEFRAG defragments one file at a time. Pages do not migrate between files. Every five minutes, DBCC INDEXDEFRAG will report to the user an estimated percentage completed. DBCC INDEXDEFRAG can be terminated at any point in the process, and any completed work is retained.

Unlike DBCC DBREINDEX (or the index building operation in general), DBCC INDEXDEFRAG is an online operation. It does not hold locks long term and thus will not block running queries or updates. A relatively unfragmented index can be defragmented faster than a new index can be built because the time to defragment is related to the amount of fragmentation. A very fragmented index might take considerably longer to defragment than to rebuild. In addition, the defragmentation is always fully logged, regardless of the database recovery model setting (see ALTER DATABASE). The defragmentation of a very fragmented index can generate more log than even a fully logged index creation. The defragmentation, however, is performed as a series of short transactions and thus does not require a large log if log backups are taken frequently or if the recovery model setting is SIMPLE.

Also, DBCC INDEXDEFRAG will not help if two indexes are interleaved on the disk because INDEXDEFRAG shuffles the pages in place. To improve the clustering of pages, rebuild the index. DBCC INDEXDEFRAG cannot correct page splits for the same reason. It essentially reorders index pages already allocated into sequential order reflective of the search key. Index pages may get out of order for a variety of reasons, such as unordered data loads, excessive insert, update, delete activity, etc.

SQL Server Books Online includes a handy piece of sample code that you can use to automate a variety of index maintenance tasks with a few modifications. The example shows a simple way to defragment all indexes in a database that have fragmented above a declared threshold. In the SQL Server Books Online topic "DBCC SHOWCONTIG", see section E, "Use DBCCSHOWCONTIG and DBCC INDEXDEFRAG to defragment the indexes in a database."

DBCC DBREINDEX

DBCC DBREINDEX can rebuild just a single specified index for a table or all indexes for a table depending on the syntax used. Similar in the approach taken to dropping and re-creating individual indexes, the DBCC DBREINDEX statement has the advantage of being able to rebuild all of the indexes for a table in one statement. This is easier than coding individual DROP INDEX and CREATE INDEX statements and a table's index or indexes can be rebuilt without knowledge of the table structure or any assigned constraints. Also, the DBCC REINDEX statement is inherently atomic. To achieve the equivalent atomicity when coding separate DROP INDEX and CREATE INDEX statements, you would have to wrap the multiple separate commands within a transaction.

DBCC DBREINDEX automatically takes advantage of more optimizations than individual DROP INDEX and CREATE INDEX statements do, especially if multiple nonclustered indexes reference a table that has a clustered index. DBCC DBREINDEX is also useful to rebuild indexes enforcing PRIMARY KEY or UNIQUE constraints without having to delete and re-create the constraints (because an index created to enforce a PRIMARY KEY or UNIQUE constraint cannot be deleted without deleting the constraint first). For example, you may want to rebuild an index on a PRIMARY KEY constraint to reestablish a given fill factor for the index.

DROP_EXISTING

Another way to rebuild or defragment an index is to drop and recreate it. Rebuilding a clustered index by deleting the old index and then re-creating the same index again is expensive because all the secondary indexes depend upon the clustering key that points to the data rows. If you simply delete the clustered index and re-create it, you may inadvertently cause all referencing nonclustered indexes to be deleted and re-created twice. The first drop/recreate occurs when you drop the clustered index. A second drop/recreate will occur when you go to re-create the clustered index.

To avoid this expense, the DROP_EXISTING clause of CREATE_INDEX allows this re-create to be performed in one step. Re-creating the index in a single step tells SQL Server that you are reorganizing an existing index and avoids the unnecessary work of deleting and re-creating the associated nonclustered indexes. This method also offers the significant advantage of using the presorted data from the existing index, thus avoiding the need to perform a sort of the data. This can significantly reduce the time and cost of re-creating the clustered index.

DROP INDEX / CREATE INDEX

The final way to perform index maintenance is simply to drop the index and then re-create it. This option is still widely practiced and might be preferable to people who are familiar with it and who have the processing window to accommodate full re-creates of all indexes on a table. The drawback to using this approach is that you must manually control events so they happen in proper sequence. When manually dropping and re-creating indexes, be sure to drop all nonclustered indexes before dropping and recreating the clustered index. Otherwise, all nonclustered indexes will automatically be recreated when you go to create the clustered index.

One advantage to manually creating nonclustered indexes is that individual nonclustered indexes can be recreated in parallel. However, your partitioning strategy may affect the resulting physical layout of the indexes. If two nonclustered indexes are rebuilt at the same time on the same file (filegroup), the pages from both indexes might be interwoven together on disk. This may cause your data to be stored in a nonsequential order. If you have multiple files (filegroups) located on different disks, you can specify separate files (filegroup) to hold the index upon creation, thus maintaining sequential contiguousness of the index pages.

The same warnings mentioned above about building indexes on pre-sorted data apply here. Clustered indexes built on sorted data do not have to perform an additional sort step, which can significantly reduce the time and processing resources needed to build the index.

FILLFACTOR and PAD_INDEX

The FILLFACTOR option provides a way to specify the percentage of open space to leave on index and data pages. The PAD_INDEX option for CREATE INDEX applies what has been specified for FILLFACTOR on the nonleaf level index pages. Without the PAD_INDEX option, FILLFACTOR affects mainly the leaf level index pages of the clustered index. It is a good idea to use the PAD_INDEX option with FILLFACTOR.

PAD_INDEX and FILLFACTOR are used to control page splitting. The optimal value to specify for FILLFACTOR depends on how much new data will be inserted within a given time frame into an 8-KB index and data page. It is important to keep in mind that SQL Server index pages typically contain many more rows than data pages because index pages contain only the data for columns associated with that index, whereas data pages hold the data for the entire row. Also bear in mind how often there will be a maintenance window that will permit the rebuilding of indexes to correct page splits, which are bound to occur. Try to rebuild the indexes only when the majority of the index and data pages have become filled with data. If a clustered index is properly selected for a table, the need to rebuild indexes will be infrequent. If the clustered index distributes data evenly so new row inserts into the table happen across all of the data pages associated with the table, the data pages will fill evenly. Overall, this provides more time before page splitting starts to occur and rebuilding the clustered index becomes necessary.

Determining the proper values to use for PAD_INDEX and FILLFACTOR requires you to make a judgment call. Your decision should be based on the performance tradeoffs between leaving a lot of open space on pages, on the one hand, and the amount of page splitting that might occur, on the other. If a small percentage for FILLFACTOR is specified, it will leave large open spaces on the index and data pages, causing SQL Server to read large numbers of partially filled pages in order to answer queries. For large read operations, SQL Server will obviously perform faster if more data is compressed onto the index and data pages. Specifying too high a FILLFACTOR may leave too little open space on pages and allow pages to overflow too quickly, causing page splitting.

Before arriving at a FILLFACTOR or PAD_INDEX value, remember that reads tend to far outnumber writes in many data warehousing environments. Periodic data loads, however, may invalidate the above statement. Many data warehouse administrators attempt to partition and structure tables/indexes to accommodate the periodic data loads they anticipate.

As a general rule of thumb, if writes are anticipated to be a substantial fraction of reads, the best approach is to specify as high a FILLFACTOR as feasible, while still leaving enough free space in each 8-KB page to avoid frequent page splitting, at least until SQL Server can reach the next available window of time needed to re-create indexes. This strategy balances I/O performance (keeping the pages as full as possible) and avoids page splitting (not letting pages overflow). If there will be no write activity into the SQL Server database, FILLFACTOR should be set at 100 percent so that all index and data pages are filled completely for maximum I/O performance.

SQL Server Tools for Analysis and Tuning

This section provides sample code to load a table with data, which is then used to illustrate the use of SQL Profiler and SQL Query Analyzer for analyzing and tuning performance.

Sample Data and Workload

To illustrate using the SQL Server performance tools, use the following example. First, the following table is constructed:

```
create table testtable
(nkey1 int identity,
col2 char(300) default 'abc',
ckey1 char(1))
```

Next, the table is loaded with 20,000 rows of test data. The data being loaded into column nkey1 lends itself to a nonclustered index. The data in column ckey1 lends itself to a clustered index and the data in col2 is merely filler to increase the size of each row by 300 bytes.

```
declare @counter int
set @counter = 1
while (@counter <= 4000)
begin
insert testtable (ckey1) values ('a')
insert testtable (ckey1) values ('b')
insert testtable (ckey1) values ('c')
insert testtable (ckey1) values ('d')
insert testtable (ckey1) values ('e')
set @counter = @counter + 1
end
```

The following queries make up the database server workload:

```
select ckey1 from testtable where ckey1 = 'a'
select nkey1 from testtable where nkey1 = 5000
select ckey1,col2 from testtable where ckey1 = 'a'
select nkey1,col2 from testtable where nkey1 = 5000
```

SQL Profiler

A common approach to performance tuning is often called mark and measure. To verify that changes made to improve performance actually do improve performance, you first need to establish a baseline or mark of the existing bad performance situation. Measure refers to establishing quantifiable ways to demonstrate that performance is improving.

SQL Profiler is a handy tool for marking and measuring. Not only can it capture activity that is taking place within your server for performance analysis; it can also playback that activity again at a later time. The playback capabilities in SQL Server provide a useful regression-testing tool. Using playback, you can conveniently determine whether actions being taken to improve performance are having the desired effect.

The playback capabilities can also simulate load or stress testing. Multiple profiler client sessions can be set up to play back simultaneously. This capability allows the administrator to easily capture activity from five concurrent users, for example, and then start ten simultaneous playbacks to emulate what the system performance might look like if there were 50 concurrent users. You can also take traces of database activity and play that activity back against a database modification under development or against a new hardware configuration being tested.

The thing to remember is that Profiler allows you to record activity that is occurring on your SQL Server databases. Profiler can be configured to watch and record one or many users executing queries against SQL Server. In addition to the SQL statements, a wide and varied amount of performance information is available for capture using the tool. Some of the performance information available for recording using Profiler includes items such as I/O statistics, CPU statistics, locking requests, Transact-SQL and RPC statistics, index and table scans, warnings and errors raised, database object create/drop, connection connect/disconnects, stored procedure operations, cursor operation, and more.

Capturing Profiler Information to Use with the Index Tuning Wizard

When used together, SQL Profiler and the Index Tuning Wizard provide a very powerful tool combination to help database administrators ensure that proper indexes are placed on tables and views. SQL Profiler records the resource consumption for queries into one of three places. Output can be directed to the .trc file, to a SQL Server table, or to the monitor. The Index Tuning Wizard can then read the captured data from either the .trc file or the SQL Server table. The Index Tuning Wizard analyzes information from the captured workload and information about the table structures, and then presents recommendations about what indexes should be created to improve performance. The Index Tuning Wizard provides a choice of automatically creating the proper indexes for the database, scheduling the index creation for a later time, or generating a Transact-SQL script that can be reviewed and executed manually.

The following steps are required for analyzing a query load:

Set up Profiler

1. Start Profiler from SQL Server Enterprise Manager by selecting **SQL Profiler** on the **Tools** menu.

2. Press CTRL+N to create a new Profiler trace. In the **Connect to SQL Server** dialog box, select the server you want to connect to.
3. Select the **SQLProfilerTuning** template from the dropdown list box.
4. Select either the **Save to file** or **Save to table** checkbox. The **Save to table** option opens the Connection dialog box, where you can save trace information to a server other than the server profiling the query. Both checkboxes can be selected if you would like to save traced activity to both. Point to a valid directory and file name if you want to save as a .trc file. Point to an existing trace table if you have already run the trace and are running it again; you can also provide a new table name if this is the first time you have captured trace activity to the table. Click **OK**.
5. Click **Run**.

Run the workload several (3-4) times

1. Start SQL Query Analyzer, either from SQL Server Enterprise Manager or from the **Start** menu.
2. Connect to SQL Server and set the current database to the database where you created the test table.
3. Enter the following queries into the query window of SQL Query Analyzer:

```
select ckey1 from testtable where ckey1 = 'a'
select nkey1 from testtable where nkey1 = 5000
select ckey1,col2 from testtable where ckey1 = 'a'
select nkey1,col2 from testtable where nkey1 = 5000
```

4. Press CTRL+E to execute the queries. Do this three or four times to generate a sample workload.

Stop SQL Profiler

- In the SQL Profiler window, click the red square to stop the Profiler trace.

Load the trace file or table into Index Tuning Wizard

1. In **SQL Profiler**, start the **Index Tuning Wizard** by selecting **Index Tuning Wizard** on the **Tools** menu. Click **Next.**
2. Select the database to be analyzed. Click **Next**.
3. Choose options of whether or not to keep existing indexes, or add indexed views.
4. Choose one of the tuning modes (**Fast**, **Medium**, or **Thorough**). Index Tuning Wizard requires less time to perform the analysis for Fast tuning mode but does a less thorough analysis – Thorough mode produces the most thorough analysis but takes the most time.
5. To locate the trace file/table that was created with SQL Profiler, select **My workload file** or the **SQL Server Trace Table**. Click **Next**.
6. In the **Select Tables to Tune** dialog box, select the tables you want to analyze and then click **Next**.
7. **Index Tuning Wizard** will analyze the traced workload and table structures to determine the proper indexes to create in the **Index Recommendations** dialog box. Click **Next**.
8. The wizard provides the choice of creating the indexes immediately, scheduling the index creation (an automated task for later execution), or creating a Transact-SQL script containing the commands to create the indexes. Select the preferred option and then click **Next**.
9. Click **Finish**.

Transact-SQL Generated by Index Tuning Wizard for the Sample Database and Workload

```
/* Created by: Index Tuning Wizard */
/* Date: 9/6/2000 */
/* Time: 4:44:34 PM */
/* Server Name: JHMILLER-AS2 */
/* Database Name: TraceDB */
/* Workload File Name: C:\Documents and Settings\jhmiller\My Documents\trace.trc */
USE [TraceDB]
go
SET QUOTED_IDENTIFIER ON
SET ARITHABORT ON
SET CONCAT_NULL_YIELDS_NULL ON
SET ANSI_NULLS ON
SET ANSI_PADDING ON
SET ANSI_WARNINGS ON
SET NUMERIC_ROUNDABORT OFF
go
DECLARE @bErrors as bit

BEGIN TRANSACTION
SET @bErrors = 0
```

```
CREATE CLUSTERED INDEX [testtable1] ON [dbo].[testtable] ([ckey1] ASC )
IF( @@error <> 0 ) SET @bErrors = 1

CREATE NONCLUSTERED INDEX [testtable2] ON [dbo].[testtable] ([nkey1] ASC )
IF( @@error <> 0 ) SET @bErrors = 1

IF( @bErrors = 0 )
COMMIT TRANSACTION
ELSE
ROLLBACK TRANSACTION
```

The indexes recommended by Index Tuning Wizard for the sample table and data are what we would expect – a clustered index on ckey1 and a nonclustered index on nkey1. There are only five unique values for ckey1 and 4000 rows of each value. Given that one of the sample queries (select ckey1, col2 from testtable where ckey1 = 'a') requires retrieval from the table based on one of the values in ckey1, it makes sense to create a clustered index on the ckey1 column. The second query (select nkey1, col2 from testtable where nkey1 = 5000) fetches one row based on the value of the column nkey1. Because nkey1 is unique and there are 20,000 rows, it makes sense to create a nonclustered index on this column.

The combination of SQL Profiler and the Index Tuning Wizard becomes very powerful in real database server environments, where many tables are used and many queries are processed. Use SQL Profiler to record a .trc file or trace table while the database server is experiencing a representative set of queries. Then load the trace into the Index Tuning Wizard to determine the proper indexes to build. Follow the prompts in the Index Tuning Wizard to automatically generate and schedule index creation jobs to run at off-peak times. You may want to run the combination of SQL Profiler and the Index Tuning Wizard regularly (perhaps weekly or monthly) to see if queries being executed on the database server have changed significantly, thus possibly requiring different indexes. Regular use of SQL Profiler and the Index Tuning Wizard together helps database administrators keep SQL Server running in top form as query workloads change and database size increase over time.

Analyzing the Information Recorded in Profiler with Query Analyzer

After the information is recorded into the SQL Server table, Query Analyzer can be used to determine which queries on the system are consuming the most resources. In this way, database administrators can concentrate on improving the queries that require the most help. Storing the trace data to a table enables you to easily select from and filter out subsets of trace data in order to identify the poorest performing queries for tuning purposes. For instance, in the example above, the column **Duration**, which is captured automatically when you use the **SQLProfiler Tuning** template, can be used to identify queries that required the greatest number of milliseconds to execute. To find the top ten percent of the longest running queries you can run a query like the following:

```
SELECT TOP 10 PERCENT *
FROM [TraceDB].[dbo].[Trace]
ORDER BY Duration DESC
```

To find the top five longest running queries you can run a query like the following:

```
SELECT TOP 5 *
FROM [TraceDB].[dbo].[Trace]
ORDER BY Duration DESC
```

To place only the rows you want to use for tuning into a separate table, consider using the following SELECT/INTO statement:

```
SELECT TOP 10 PERCENT *
INTO TuningTable
FROM [TraceDB].[dbo].[Trace]
ORDER BY Duration DESC
```

The **SQLProfiler Tuning** template, mentioned above, is simply a suggested set of preselected columns and filter settings recommended for tuning purposes. You may find that you want to capture more information. It is certainly possible for you to create your own custom tuning templates by simply opening one of the presupplied templates and saving it under a different name. Many events can be captured, including I/O statistics, locking information, and much more.

SQL Query Analyzer

You can use SQL Query Analyzer to tune queries. This tool provides a number of mechanisms such as Statistics I/O and the execution plans you can use to troubleshoot problem queries.

Statistics I/O

SQL Query Analyzer provides an option that you can use to obtain information about the I/O consumption for a query that you execute in Query Analyzer. To set this option, in Query Analyzer, select **Current Connection Properties** on the **Query** menu to display the **Current Connection Properties** dialog box. Select the **Set statistics** I/O checkbox and close the dialog box. Next, execute a query and select the **Message** tab in the results pane to see the I/O

statistics.

For example, the following query on the sample data created earlier in the SQL Profiler section returns the following I/O information on the messages tab when the **Set statistics IO** option is selected:

```
select ckey1, col2 from testtable where ckey1 = 'a'
Table 'testtable'. Scan count 1, logical reads 800, physical reads 62, read-ahead reads 760.
```

Using statistics I/O is a great way to monitor the effect of query tuning. For example, create the indexes that Index Tuning Wizard suggested for the sample data and then run the query again.

```
select ckey1, col2 from testtable where ckey1 = 'a'
Table 'testtable'. Scan count 1, logical reads 164, physical reads 4, read-ahead reads 162.
```

Notice that the number of logical and physical reads is significantly lower when an index is available.

Execution Plan

Graphical execution plans can be used to focus attention on problematic SQL queries by displaying detailed information on what the query optimizer is doing.

An estimated execution plan for a query can be displayed in the Results pane of Query Analyzer by executing an SQL query with CTRL+L or by selecting **Display Estimated Execution Plan** on the **Query** menu. Icons indicate the operations that the query optimizer would have performed if it had executed the query. Arrows indicate the direction of data flow for the query. Details about each operation can be displayed by hovering the mouse pointer over the operation icon. The approximate cost of each step of the operation is also noted beneath each operation icon. This label allows you to quickly zero in on which operation is most expensive in the query.

You can also see the actual execution plan for a query by selecting **Show Execution Plan** on the **Query** menu and then executing the query. In contrast to the **Display Estimated Execution Plan** option, **Show Execution plan** executes the query before displaying the actual execution plan used for the query.

A text version of an execution plan can be created by selecting **Current Connection Properties** on the **Query** menu and then checking the **Set showplan_text** option in the dialog box. The execution plan will be displayed as text in the results tab when the query is executed.

Execution plan options can also be set within the query by executing either of the following commands:

```
set showplan_all on
go
set showplan_text on
go
```

SET SHOWPLAN_ALL is intended to be used by applications designed to read its output. Use SET SHOWPLAN_TEXT to return readable output for Microsoft MS-DOS® applications, such as the **osql** utility.

SET SHOWPLAN_TEXT and SET SHOWPLAN_ALL return information as a set of textual rows that form a hierarchical tree representing the steps taken by the SQL Server query processor as it executes each statement. Each statement reflected in the output contains a single row with the text of the statement, followed by several rows with the details of the execution steps.

Examples of Execution Plan Output

Using the example queries defined earlier and "set showplan_text on" executed in Query Analyzer provides these results.

Query 1

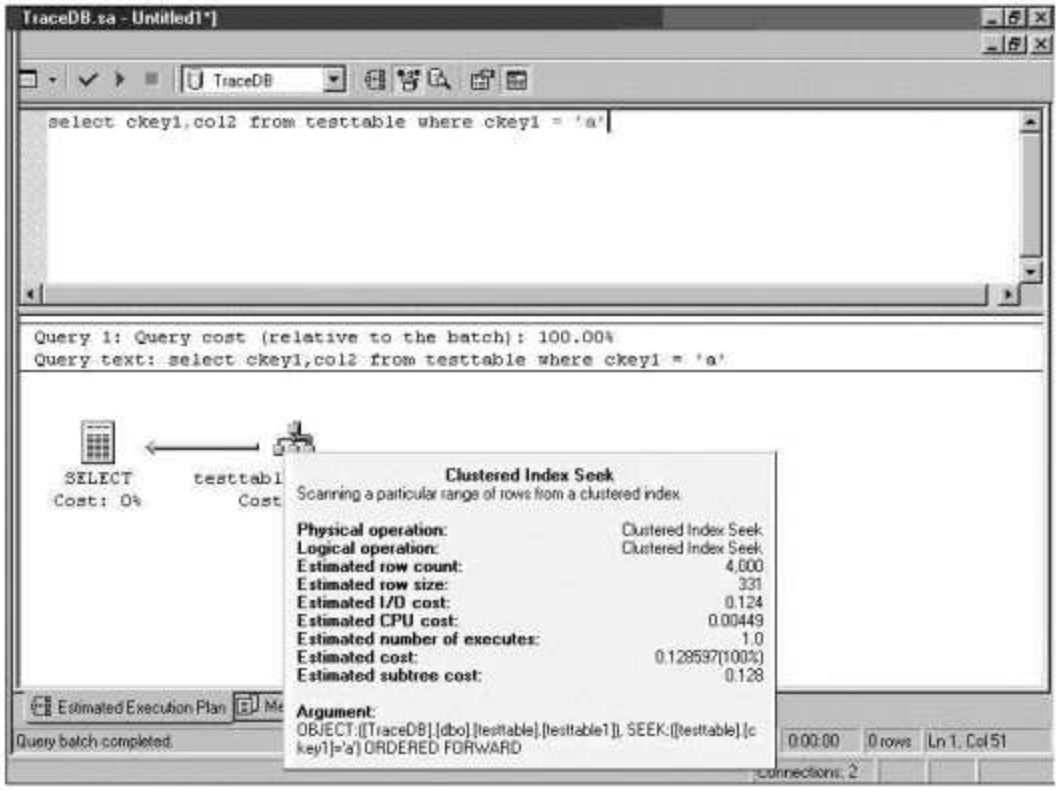
```
select ckey1,col2 from testtable where ckey1 = 'a'
```

Text-based execution plan output

```
|--Clustered Index Seek (OBJECT:([TraceDB].[dbo].[testtable].[testtable1]), SEEK:([testtable].[ckey1]
='a') ORDERED FORWARD)
```

Equivalent graphical execution plan output

The following illustration shows the graphical execution plan for query 1.



If your browser does not support inline frames, click here to view on a separate page.

The execution plan takes advantage of the clustered index on column ckey1 to resolve the query, as indicated by **Clustered Index Seek**.

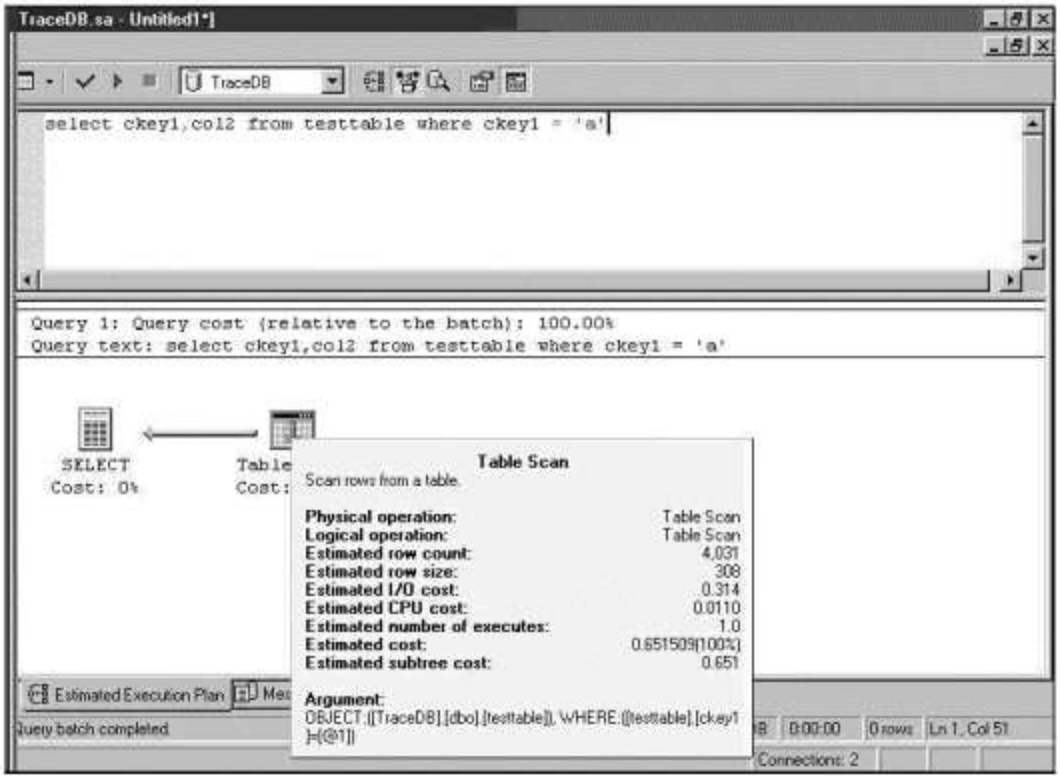
If the clustered index is removed from the table and the same query is executed again, the query reverts to using a table scan. The following graphical execution plan indicates the change in behavior.

Text-based execution plan output

```
|--Table Scan(OBJECT:([TraceDB].[dbo].[testtable]), WHERE:([testtable].[ckey1]=[@1]))
```

Equivalent graphical execution plan output

The following illustration shows the graphical execution plan for query 1.



If your browser does not support inline frames, click here to view on a separate page.

This execution plan uses a table scan to resolve query 1. Table scans are the most efficient way to retrieve information from small tables. But on larger tables, table scans indicated by an execution plan are a warning that the table may need better indexes or that the existing indexes need to have their statistics updated. Statistics can be updated on a table or an index using the UPDATE STATISTICS command. SQL Server automatically updates indexes if the heuristic pages get too far out of synch with the underlying index values. An example would be if you deleted all the rows containing a ckey1 value = "b" from your testtable and then ran queries without first updating the statistics. It is a good idea to allow SQL Server to automatically maintain index statistics because it helps guarantee that queries will always have good index statistics to work with. SQL Server will not auto update statistics if you set the AUTO_UPDATE_STATISTICS database options to OFF using the ALTER DATABASE statement.

Query 2

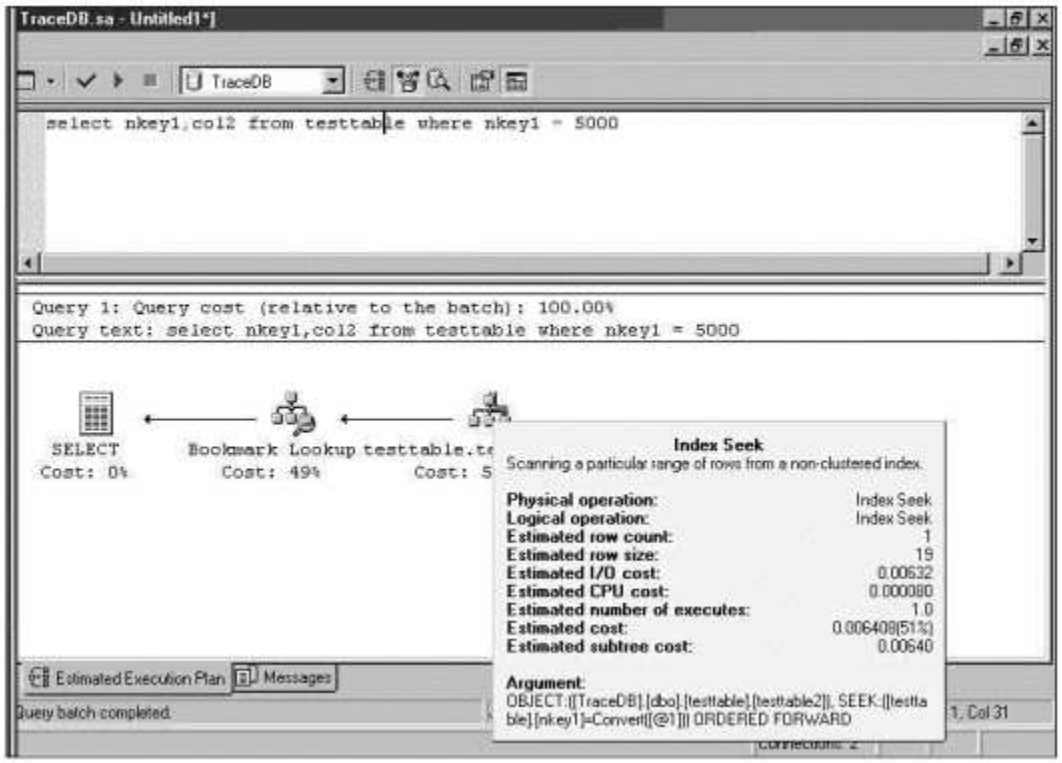
```
select nkey1,col2 from testtable where nkey1 = 5000
```

Text-based execution plan output

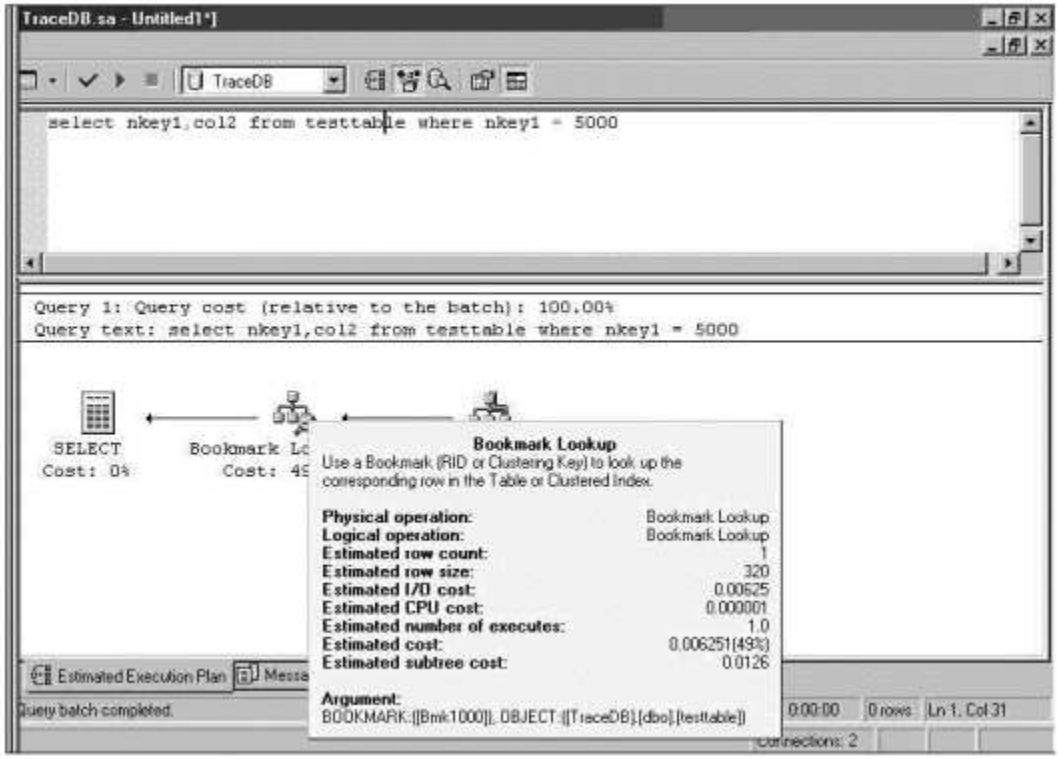
```
--Bookmark Lookup(BOOKMARK:([Bmk1000]),
OBJECT:([TraceDB].[dbo].[testtable]))
|--Index Seek(OBJECT:([TraceDB].[dbo].[testtable].[testtable2]),
SEEK:([testtable].[nkey1]=Convert([@1])) ORDERED FORWARD)
```

Equivalent graphical execution plan output

The following two illustrations show the graphical execution plan for query 2.



If your browser does not support inline frames, click here to view on a separate page.



If your browser does not support inline frames, click here to view on a separate page.

The execution plan for query 2 uses the nonclustered index on the column nkey1. This is indicated by the Index Seek operation on the column nkey1. The Bookmark Lookup operation indicates that SQL Server needed to perform a pointer jump from the index page to the data page of the table to retrieve the requested data. The pointer jump was required because the query asked for the column col2, which was not a column contained within the nonclustered index.

Query 3

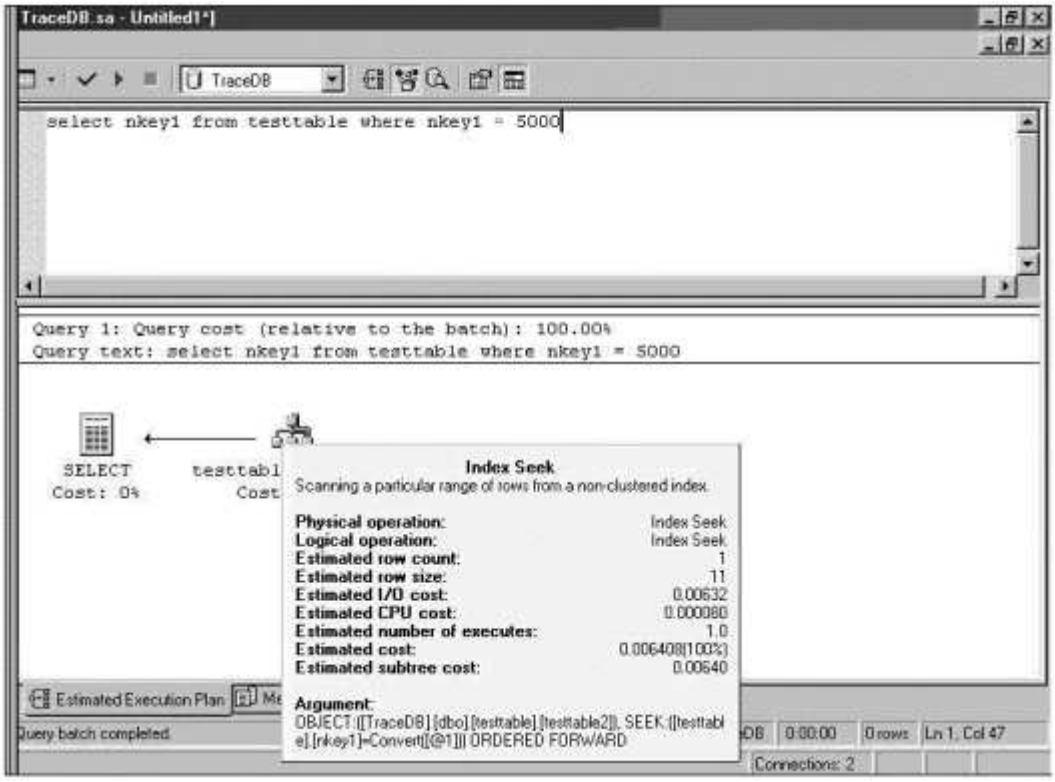
```
select nkey1 from testtable where nkey1 = 5000
```

Text-based execution plan output

```
|--Index Seek(OBJECT:([TraceDB].[dbo].[testtable].[testtable2]), SEEK:([testtable].[nkey1]=Convert
([@1])) ORDERED FORWARD)
```

Equivalent graphical execution plan output

The following illustration shows the graphical execution plan for query 3.



If your browser does not support inline frames, click here to view on a separate page.

The execution plan for query 3 uses the nonclustered index on nkey1 as a covering index. Note that no Bookmark Lookup operation was needed for this query. This is because all of the information required for the query (both SELECT and WHERE clauses) is provided by the nonclustered index. This means that no pointer jumps to the data pages are required from the nonclustered index pages. I/O is reduced in comparison to the case where a bookmark lookup was required.

System Monitoring

System Monitor (Performance Monitor in Windows NT 4) provides a wealth of information about what Windows and SQL Server operations are taking place during database server execution.

In System Monitor graph mode, take note of the **Max** and **Min** values. Don't put too much emphasis on the average. Because heavily polarized data points can distort the average, be careful of overemphasizing the average. Study the graph shape and compare to **Min** and **Max** to get an accurate understanding of the behavior. Use the BACKSPACE key to highlight counters with a white line.

It is possible to use System Monitor to log all available Windows and SQL Server system monitor objects/counters in a log file, while at the same time looking at System Monitor interactively (chart mode). The setting of sampling interval determines how quickly the logfile grows in size. Logfiles can get very large, very fast (for example, 100 megabytes in one hour with all counters turned on and a sampling interval of 15 seconds). It is hoped that the test server has enough free gigabytes to store these types of files. If conserving space is important, however, try running with a large log interval so System Monitor does not sample the system as often. Try 30 or 60 seconds. This way all of the counters are resampled with reasonable frequency but a smaller logfile size is maintained.

System Monitor also consumes a small amount of CPU and disk I/O resources. If a system does not have much disk I/O and/or CPU to spare, consider running System Monitor from another machine to monitor SQL Server over the network. When monitoring over the network, use graph mode only — it tends to be more efficient to log performance monitoring

information locally on the SQL Server instead of sending the information across a local area network. If you must log over the network, reduce the logging to only the most critical counters.

It is a good practice to log all counters available during performance test runs into a file for later analysis. That way any counter can be examined further at a later time. Configure System Monitor to log all counters into a logfile and at the same time monitor the most interesting counters in one of the other modes, like graph mode. This way, all of the information is recorded but the most interesting counters are presented in an uncluttered System Monitor graph while the performance run is taking place.

Setting up a System Monitor Session to be Logged

1. From the Windows 2000 **Start** menu point to **Programs**, point to **Administrative Tools**, and then click **Performance** to open the System Monitor.
2. Double-click **Performance Logs and Alerts**, and then click **Counter Logs**.
3. Any existing logs will be listed in the details pane. A green icon indicates that a log is running; a red icon indicates that a log has been stopped.
4. Right-click a blank area of the details pane, and click **New Log Settings**.
5. In **Name**, type the name of the log, and then click **OK**.
6. On the **General** tab, click **Add**. Select the counters you want to log. This is where you decide which SQL Server counters to monitor during the session.
7. If you want to change the default file, make the changes on the **Log Files** tab.
8. Logged sessions can be set to run automatically for predefined time periods. To do this, you modify the schedule information on the **Schedule** tab.

   Note To save the counter settings for a log file, right-click the file in the details pane and click **Save Settings As**. You can then specify an .htm file in which to save the settings. To reuse the saved settings for a new log, right-click the details pane, and then click **New Log Settings From**.

Starting a Logged Monitoring Session

1. From the Windows 2000 **Start** menu, point to **Programs**, point to **Administrative Tools**, and then select **Performance** to open the System Monitor.
2. Double-click **Performance Logs and Alerts**, and then click **Counter Logs**.
3. Right-click the **Counter Log** to be run and select start.
4. Any existing logs will be listed in the details pane. A green icon indicates that a log is running; a red icon indicates that a log has been stopped.

Stopping a Logged Monitoring Session

1. From the Windows 2000 **Start** menu point to **Programs**, point to **Administrative Tools**, and then select **Performance** to open the System Monitor.
2. Double-click **Performance Logs and Alerts**, and then click **Counter Logs**.
3. Right-click the **Counter Log** to be run and select stop.

Loading data from a Logged Monitoring Session into System Monitor for analyzing

1. From the Windows 2000 **Start** menu point to **Programs**, point to **Administrative Tools**, and then select **Performance** to open the System Monitor.
2. Click **System Monitor**.
3. Right-click the System Monitor details pane and click **Properties**.
4. Click the **Source** tab.
5. Under **Data Source**, click **Log File**, and type the path to the file or click **Browse** to browse for the log file you want.
6. Click **Time Range**. To specify the time range in the log file that you want to view, drag the bar or its handles for the appropriate starting and ending times.
7. Click the **Data** tab and click **Add** to open the **Add Counters** dialog box. The counters you selected during log configuration are shown. You can include all or some of these in your graph.

How to relate System Monitor logged events back to a point in time

- From within the System Monitor session, right-click the System Monitor details pane and click **Properties**. A time range and a slider bar lets you position the begin, current, and end times to be viewed in your graph.

Key Performance Counters to Watch

Several performance counters provide information about the following areas of interest: memory, paging, processors,

I/O, and disk activity.

Monitoring Memory

By default, SQL Server changes its memory requirements dynamically, based on available system resources. If SQL Server needs more memory, it queries the operating system to determine whether free physical memory is available and uses the available memory. If SQL Server does not need the memory currently allocated to it, it releases the memory to the operating system. However, the option to dynamically use memory can be overridden using the **min server memory**, **max server memory**, and **set working set size** server configuration options. For more information, see SQL Server Books Online.

To monitor the amount of memory being used by SQL Server, examine the following performance counters:

- Process: Working Set
- SQL Server: Buffer Manager: Buffer Cache Hit Ratio
- SQL Server: Buffer Manager: Total Pages
- SQL Server: Memory Manager: Total Server Memory (KB)

The **Working Set** counter shows the amount of memory used by a process. If this number is consistently below the amount of memory SQL Server is configured to use (set by the **min server memory** and **max server memory** server options), SQL Server is configured for more memory than it needs. Otherwise, adjust the size of the working set using the **set working set size** server option.

The **Buffer Cache Hit Ratio** counter is application specific; however, a rate of 90 percent or higher is desirable. Add more memory until the value is consistently greater than 90 percent, indicating that more than 90 percent of all requests for data were satisfied from the data cache.

If the **Total Server Memory (KB)** counter is consistently high compared to the amount of physical memory in the computer, more memory may be required.

Hard Paging

If **Memory: Pages/sec** is greater than zero or **Memory: Page Reads/sec** is greater than five, Windows is using disk to resolve memory references (hard page fault). This costs disk I/O + CPU resources. **Memory: Pages/sec** is a good indicator of the amount of paging that Windows is performing and the adequacy of the database server's current RAM configuration. A subset of the hard paging information in System Monitor is the number of times per second Windows had to read from the paging file to resolve memory references, which is represented by **Memory: Pages Reads/sec**. If **Memory: Pages Reads/sec** > 5, this is bad for performance.

Automatic SQL Server memory tuning will try to adjust SQL Server memory utilization dynamically in order to avoid paging. A small number of pages read per second is normal, but excessive paging requires corrective action.

If SQL Server is automatically tuning memory, adding more RAM or removing other applications from the database server are two options to help bring **Memory: Pages/sec** to a reasonable level.

If SQL Server memory is manually configured on the database server, it may be necessary to reduce memory given to SQL Server, remove other applications from the database server, or add more RAM to the database server.

Keeping **Memory: Pages/sec** at or close to zero helps database server performance. It means Windows and all its applications (this includes SQL Server) are not going to the paging file to satisfy any data in memory requests, so the amount of RAM on the server is sufficient. If **Pages/sec** is greater than zero by a small amount, this is acceptable, but remember that a relatively high performance penalty (disk I/O) is paid every time data is retrieved from the paging file rather than RAM.

It is useful to compare **Memory: Pages Input/sec** to **Logical Disk: Disk Reads/sec** across all drives associated with the Windows paging file, and **Memory: Page Output/sec** to **Logical Disk: Disk Writes/sec** across all drives associated with the Windows paging file, because they provide a measure of how much disk I/O is strictly related to paging rather than other applications (that is, SQL Server). Another easy way to isolate paging file I/O activity is to make sure that the paging file is located on a separate set of drives from all other SQL Server files. Separating the paging file away from the SQL Server files can also help disk I/O performance because it allows disk I/O associated with paging to be performed in parallel to disk I/O associated with SQL Server.

Soft Paging

If **Memory: Pages Faults/sec** is greater than zero, Windows is paging, but includes both hard and soft paging within the counter. In the previous section, we discussed hard paging. Soft paging means that application(s) on the database server are requesting memory pages still inside RAM but outside of Windows **Working Set**. **Memory: Page Faults/sec** is helpful for deriving the amount of soft paging that is occurring. There is no counter called Soft Faults per second. Instead, use this computation to calculate the number of soft faults happening per second: **Memory: Pages Faults/sec** - **Memory: Pages Input/sec** = **Soft Page Fault/sec**.

To determine if SQL Server, rather than another process, is causing excessive paging, monitor the **Process: Page Faults/sec** counter for the SQL Server process and note whether the number of page faults per second for the Sqlservr.exe instance in question is similar to the number of **Memory: Pages/sec**.

Soft faults generally are not as bad as hard faults for performance because they consume CPU resources. Hard faults consume disk I/O resources. The best environment for performance is to have no faulting of any kind.

Note When SQL Server accesses all of its data cache pages for the first time, the first access to each page will cause a soft fault. Do not be concerned with initial soft faulting occurring when SQL Server first starts up and the data cache is first being exercised.

Monitoring processors

Your goal should be to keep all of the processors that are allocated to the server busy enough to maximize performance, but not so busy that processor bottlenecks occur. The performance tuning challenge is that if CPU is not the bottleneck, something else is (a primary candidate is the disk subsystem), so CPU capacity is being wasted. CPU is usually the hardest resource to expand (above some configuration specific level, such as four or eight on many current systems), so it should be seen as a good sign that CPU utilization is more than 95 percent on busy systems. At the same time, the response time of transactions should be monitored to ensure they are within reason; if not, CPU usage greater than 95 percent may simply mean that the workload is too much for the available CPU resources and either CPU resources have to be increased or workload has to be reduced or tuned.

Look at the System Monitor counter **Processor: % Processor Time** to make sure all processors are consistently below 95 percent utilization on each CPU. **System:Processor Queue Length** is the processor queue for all CPUs on a Windows system. If **System: Processor Queue Length** is greater than two for each CPU, it indicates a CPU bottleneck. When a CPU bottleneck is detected, it is necessary to either add processors to the server or reduce the workload on the system. Reducing workload can be accomplished by query tuning or improving indexes to reduce I/O and, subsequently, CPU usage.

Another System Monitor counter to watch when a CPU bottleneck is suspected is **System: Context Switches/sec** because it indicates the number of times per second that Windows and SQL Server had to change from executing on one thread to executing on another. This costs CPU resources. Context switching is a normal component of a multithreaded, multiprocessor environment, but excessive context switching can degrade system performance. The approach to take is to only worry about context switching if there is processor queuing.

If processor queuing is observed, use the level of context switching as a gauge when performance tuning SQL Server. If context switching seems to be a contributor, there are two approaches you might want to consider: using the **affinity mask** option, and using fiber-based scheduling.

Use the **affinity mask** option to increase performance on symmetric multiprocessor (SMP) systems (with more than four microprocessors) operating under heavy load. You can associate a thread with a specific processor and specify which processors SQL Server will use. You can also exclude SQL Server activity from using certain processors using an **affinity mask** option setting. Before you change the setting of **affinity mask**, keep in mind that Windows assigns deferred process call (DPC) activity associated with NICs to the highest numbered processor in the system. In systems with more than one NIC installed and active, each additional card's activity is assigned to the next highest numbered processor. For example, an eight-processor system with two NICs has DPCs for each NIC assigned to processor 7 and to processor 6 (0-based counting is used). When using the **lightweight pooling** option, SQL Server switches to a fiber-based scheduling model rather than the default thread-based scheduling model. You can think of fibers as essentially lightweight threads. Use the command `sp_configure 'lightweight pooling',1` to enable fiber-based scheduling.

Watch processor queuing and context switching to monitor the effect of setting values for both **affinity mask** and **lightweight pooling**. In some situations, these settings can make performance worse instead of better. Also, they generally do not yield much benefit unless your system has four or more processors. DBCC SQLPERF (THREADS) provides more information about I/O, memory, and CPU usage mapped back to spids. Execute the following SQL query to take a survey of current top consumers of CPU time:

```
select * from master.sysprocesses order by cpu desc.
```

Monitoring Processor Queue Length

If **System: Processor Queue Length** is greater than two, this means the server's processors are receiving more work requests than they can handle as a collective group. Therefore, Windows needs to place these requests in a queue.

Some processor queuing is an indicator of good overall SQL Server I/O performance. If there is no processor queuing and if CPU utilization is low, it may be an indication that there is a performance bottleneck somewhere else in the system, the most likely candidate being the disk subsystem. Having a reasonable amount of work in the processor queue means that the CPUs are not idle and the rest of the system is keeping pace with the CPUs.

A general rule of thumb for a good processor queue number is to multiply the number of CPUs on the database server by two.

Processor queuing significantly above this calculation needs to be investigated and may indicate that your server is experiencing a CPU bottleneck. Excessive processor queuing costs query execution time. Several different activities could be contributing to processor queuing. Eliminating hard and soft paging will help save CPU resources. Other methodologies that help reduce processor queuing include SQL query tuning, picking better indexes to reduce disk I/O (and, hence, CPU), or adding more CPUs (processors) to the system.

Monitoring I/O

Disk Write Bytes/sec and **Disk Read Bytes/sec** counters provide an idea of the data throughput in terms of bytes per second per logical or physical drive. Weigh these numbers carefully along with **Disk Reads/sec** and **Disk Writes/sec**. Do not let a low amount of bytes per second lead you to believe that the disk I/O subsystem is not busy.

Monitor the **Disk Queue Length** for all drives associated with SQL Server files and determine which files are associated with excessive disk queuing.

If System Monitor indicates that some drives are not as busy as others, there is the opportunity to move SQL Server files from drives that are bottlenecking to drives that are not as busy. This will help spread disk I/O activity more evenly across hard drives. If one large drive pool is being used for SQL Server files, the resolution to disk queuing is to make the I/O capacity of the pool bigger by adding more physical drives to the pool.

Disk queuing may be an indication that one SCSI channel is being saturated with I/O requests. System Monitor cannot directly determine if this is the case. Storage vendors generally offer additional tools to help monitor the amount of I/O being serviced by a RAID controller and whether the controller is queuing I/O requests. This is more likely to occur if many disk drives (ten or more) are attached to the SCSI channel and they are all performing I/O at full speed. In this case, the solution is to connect half of the disk drives to another SCSI channel or RAID controller to balance that I/O. Typically, rebalancing drives across SCSI channels requires a rebuild of the RAID arrays and full backup/restore of the SQL Server database files.

Percent Disk Time

In System Monitor, the **PhysicalDisk: % Disk Time** and **LogicalDisk: % Disk Time** counters monitor the percentage of time that the disk is busy with read/write activity. If the **% Disk Time** counter is high (more than 90 percent), check the **Current Disk Queue Length** counter to see how many system requests are waiting for disk access. The number of waiting I/O requests should be sustained at no more than 1.5 to 2 times the number of spindles making up the physical disk. Most disks have one spindle, although redundant array of inexpensive disks (RAID) devices usually have more. A hardware RAID device appears as one physical disk in System Monitor; RAID devices created through software appear as multiple instances.

Disk Queue Length

It is important to monitor for excessively long disk queues.

To monitor disk queue length, you will need to observe several System Monitor disk counters. To enable these counters, run the command `diskperf -y` from the Windows 2000 or Windows NT command window and restart the machine.

Physical hard drives that are experiencing disk queuing will hold back disk I/O requests while they catch up on I/O processing. SQL Server response time will be degraded for these drives. This costs query execution time.

If you use RAID, it is necessary to know how many physical hard drives are associated with each drive array that Windows sees as a single physical drive, in order to calculate disk queuing for each physical drive. Ask a hardware expert to explain the SCSI channel and physical drive distribution in order to understand how SQL Server data is held by each physical drive and how much SQL Server data is distributed on each SCSI channel.

There are several choices for looking at disk queuing through System Monitor. Logical disk counters are associated with the logical drive letters assigned through Disk Administrator, whereas physical disk counters are associated with what Disk Administrator sees as a single physical disk device. Note that what appears to Disk Administrator as a single physical device may either be a single hard drive or a RAID array, which consists of several hard drives. **Current Disk Queue Length** is an instantaneous measure of disk queuing whereas **Average Disk Queue Length** averages the disk queuing measurement over the sampling period. Take note if one of the following conditions is indicated:

Logical Disk: Avg. Disk Queue Length > 2

Physical Disk: Avg. Disk Queue Length > 2

Logical Disk: Current Disk Queue Length > 2

Physical Disk: Current Disk Queue Length > 2

These recommended measurements are for each physical hard drive. If a RAID array is associated with a disk queue measurement, the measurement needs to be divided by the number of physical hard drives in the RAID array to determine the disk queuing per physical hard drive.

Note On physical hard drives or RAID arrays that hold SQL Server log files, disk queuing is not a useful measure because the log manager does not queue more than a single I/O request to SQL Server logfile(s).

Understanding SQL Server Internals

Understanding some of the internals of SQL Server 2000 can assist you in managing the performance of your databases.

Worker Threads

SQL Server maintains of pool of Windows threads that are used to service batches of SQL Server commands being submitted to the database server. The total number of these threads (referred to in SQL Server terminology as worker threads) available to service all incoming command batches is dictated by the setting for the **sp_configure** option **max worker threads**. If the number of connections actively submitting batches is greater than the number specified for **max worker threads**, worker threads will be shared among connections actively submitting batches. The default of 255 will work well for many installations. Note that the majority of connections spend most of their time waiting for batches to be received from the client.

Worker threads take on most of the responsibility of writing out dirty 8-KB pages from the SQL Server buffer cache. Worker threads schedule their I/O operations asynchronously for maximum performance.

Lazy Writer

The lazy writer is a SQL Server system process that functions within the buffer manager. The lazy writer flushes out batches of dirty, aged buffers (buffers containing changes that must be written back to disk before the buffer can be reused for a different page) and makes them available to user processes. This activity helps to produce and maintain available free buffers, which are 8-KB data cache pages empty of data and available for reuse. As the lazy writer flushes each 8-KB cache buffer to disk, the identity of the cache page is initialized so other data may be written into the free buffer. The lazy writer minimizes the impact of this activity on other SQL Server operations by working during periods of low disk I/O.

SQL Server automatically configures and manages the level of free buffers. The performance counter **SQL Server: Buffer Manager: Lazy Writes/sec** indicates the number of 8-KB pages being physically written out to disk. Monitor **SQL Server: Buffer Manager: Free Pages** to see if this value dips. Optimally, the lazy writer keeps this counter level throughout SQL Server operations, which means the lazy writer is keeping up with the user demand for free buffers. If the value of System Monitor object **SQL Server: Buffer Manager: Free Pages** reaches zero, there were times when the user load demanded a higher level of free buffers than the lazy writer was able to provide.

If the lazy writer is having problems keeping the free buffer steady, or at least above zero, it could mean the disk subsystem is not able to provide sufficient disk I/O performance. Compare drops in free buffer level to disk queuing to confirm this. The solution is to add more physical disk drives to the database server disk subsystem in order to provide more disk I/O processing power.

Monitor the current level of disk queuing in System Monitor by looking at the performance counters **Average Disk Queue Length** or **Current Disk Queue Length** for logical or physical disks, and ensure the disk queue is less than 2 for each physical drive associated with any SQL Server activity. For database servers that employ hardware RAID controllers and disk arrays, remember to divide the number reported by Logical/Physical Disk counters by the number of actual hard drives associated with that logical drive letter or physical hard drive number (as reported by Disk Administrator), because Windows and SQL Server are unaware of the actual number of physical hard drives attached to a RAID controller. It is important to be aware of the number of drives associated with the RAID array controller in order to properly interpret the disk queue numbers that System Monitor is reporting.

For more information, search for the strings "freeing and writing buffer pages" and "write-ahead transaction log" in SQL Server Books Online.

Checkpoint

Periodically, each instance of SQL Server ensures that all dirty log and data pages are flushed to disk. This is called a checkpoint. Checkpoints reduce the time and resources needed to recover from a failure when an instance of SQL Server is restarted. During a checkpoint, dirty pages (buffer cache pages that have been modified since being brought into the buffer cache) are written to the SQL Server data files. A buffer written to disk at a checkpoint still contains the page and users can read or update it without rereading it from disk, which is not the case for free buffers created by the lazy writer.

Checkpoint logic attempts to let worker threads and the lazy writer do the majority of the work writing out dirty pages. Checkpoint logic does this by trying an extra checkpoint wait before writing out a dirty page if possible. This provides the worker threads and the lazy writer more time to write out the dirty pages. The conditions under which this extra wait time for a dirty page occurs is detailed in SQL Server Books Online in the topic "Checkpoints and the Active Portion of the Log." The main idea to remember is that checkpoint logic attempts to even out SQL Server disk I/O activity over a longer time period with this extra checkpoint wait.

For more efficient checkpoint operations when there are a large number of pages to flush out of cache, SQL Server sorts the data pages to be flushed in the order the pages appear on disk. This helps to minimize disk arm movement during cache flush and takes advantage of sequential disk I/O where possible. The checkpoint process also submits 8-KB disk I/O requests asynchronously to the disk subsystem. This allows SQL Server to finish submitting required disk I/O requests faster because the checkpoint process doesn't wait for the disk subsystem to report back that the data has

been actually written to disk.

It is important to watch disk queuing on hard drives associated with SQL Server data files to determine if SQL Server is sending more disk I/O requests than the disk(s) can handle; if this is true, more disk I/O capacity must be added to the disk subsystem so it can handle the load.

Log Manager

Like all other major RDBMS products, SQL Server ensures that all write activity (insert, update, and delete) performed on the database will not be lost if something were to interrupt SQL Server's online status, such as power failure, disk drive failure, fire in the data center, and so on. The SQL Server logging process helps guarantee recoverability. Before any implicit (single SQL query) or explicit transaction (defined transaction that issues a BEGIN TRAN/COMMIT, or ROLLBACK command sequence) can be completed, the log manager must receive a signal from the disk subsystem that all data changes associated with that transaction have been written successfully to the associated log file. This rule guarantees that if SQL Server is abruptly shut down for whatever reason and the transactions written into the data cache are not yet flushed to the data files by the checkpoint and lazy writer, the transaction log can be read and reapplied in SQL Server upon startup. Reading the transaction log and applying the transactions to SQL Server data after a server stoppage is referred to as recovery.

Because SQL Server must wait for the disk subsystem to complete I/O to SQL Server log files as each transaction is completed, it is important that the disks containing SQL Server log files have sufficient disk I/O handling capacity for the anticipated transaction load.

The method of watching out for disk queuing associated with SQL Server log files is different from SQL Server database files. Use the System Monitor counters **SQL Server: Databases <database instance>: Log Flush Waits Times** and **SQL Server: Databases <database instance>: Log Flush Waits/sec** to see if there are log writer requests waiting on the disk subsystem for completion.

A caching controller provides the highest performance, but should not be used for disks that contain log files unless the controller guarantees that data entrusted to it will be written to disk eventually, even if the power fails. For more information on caching controllers, refer to the section in this document titled "Effect of On-Board Cache of Hardware RAID Controllers."

Read-Ahead Management

SQL Server 2000 provides automatic management for reading large sequential reads for activities such as table scans. Read-ahead management is completely self-configuring and self-tuning, and is tightly integrated with the operations of the SQL Server query processor. Read-ahead management is used for large table scans, large index range scans, probes into clustered and nonclustered index binary trees, and other situations. This is because read-aheads occur with 64-KB I/Os, which provide higher disk throughput potential for the disk subsystem than do 8-KB I/Os. When it is necessary to retrieve a large amount of data, SQL Server uses read-ahead to maximize throughput.

SQL Server uses a simple and efficient Index Allocation Map (IAM) storage structure that supports read-ahead management. The IAM is the SQL Server mechanism for recording the location of extents – each 64 KB extent contains eight pages of data or index information. Each IAM page is an 8-KB page that contains tightly packed (bitmapped) information about which extents contain required data. The compact nature of IAM pages makes them fast to read, and more regularly used IAM pages can be maintained in buffer cache.

Read-ahead management can construct multiple sequential read requests by combining query information from the query processor with information about the location of all extents that need to be read from the IAM page(s). Sequential 64-KB disk reads provide extremely good disk I/O performance. The **SQL Server: Buffer Manager: Read-Ahead Pages/sec** performance counter provides information about the effectiveness and efficiency of read-ahead management.

SQL Server 2000 Enterprise Edition dynamically adjusts the maximum number of read ahead pages based on the amount of memory present. For all other editions of SQL Server 2000 the value is fixed. Another advance in SQL Server 2000 Enterprise Edition is commonly called merry-go-round scan, which allows multiple tasks to share full table scans. If the execution plan of an SQL statement calls for a scan of the data pages in a table, and if the relational database engine detects that the table is already being scanned for another execution plan, the database engine joins the second scan to the first at the current location of the second scan. The database engine reads each page once and passes the rows from each page to both execution plans. This continues until the end of the table is reached. At that point, the first execution plan has the complete results of a scan, but the second execution plan must still retrieve the data pages that occur before the point at which it joined the in-progress scan. The scan for second execution plan then wraps back to the first data page of the table and scans forward to the point at which it joined the first scan. Any number of scans can be combined in this way; the database engine will keep looping through the data pages until it has completed all the scans.

One caveat about read-ahead management is that too much read-ahead can be detrimental overall to performance because it can fill cache with unneeded pages, using I/O and CPU that could have been used for other purposes. The solution is a general performance tuning goal to tune all SQL queries so a minimal number of pages are brought into buffer cache. This includes making sure you have the right indexes in place and are using them. Use clustered indexes for efficient range scanning and define nonclustered indexes to help quickly locate single rows or smaller rowsets. For

example, if you plan to have only one index in a table and that index is for the purposes of fetching single rows or smaller rowsets, you should make the index clustered. Clustered indexes are nominally faster than nonclustered indexes.

Miscellaneous Performance Topics

Database Design Using Star and Snowflake Schemas

Data warehouses use dimensional modeling to organize data for the purpose of analysis. Dimensional modeling produces star and snowflake schemas, which also provide performance efficiency for the massive data read operations that are frequently performed in data warehousing. High-volume data (often hundreds of millions of rows) is stored in a fact table that has very short rows, which minimizes storage requirements and query time. Attributes of business facts are denormalized into dimension tables to minimize the number of table joins when retrieving data.

For a discussion of database design for data warehouses, see the chapter "Data Warehouse Design Considerations," in the *Microsoft SQL Server 2000 Resource Kit*.

SQL to Avoid, If Possible

Using inequality operators in SQL queries will force databases to use table scans to evaluate the inequalities. This generates high I/O if these queries regularly run against very large tables. WHERE clauses that contain the "NOT" operators (`!=, <>, !<, !>`), such as `WHERE <column_name> != some_value` will generate high I/O.

If these types of queries need to be run, try to restructure the queries to eliminate the NOT keyword. For example:

Instead of:

```
select * from tableA where col1 != "value"
```

Try using:

```
select * from tableA where col1 < "value" and col1 > "value"
```

Reduce Rowset Size and Communications Overhead

Database programmers who work in SQL work with easy-to-use interfaces like the Microsoft ActiveX® Data Objects (ADO), Remote Data Objects (RDO) and Data Access Objects (DAO) database APIs need to consider the result sets they are building. ADO/RDO/DAO provide programmers with great database development interfaces that allow rich SQL rowset functionality without requiring a lot of SQL programming experience. But this comes at a cost. Programmers can avoid performance problems if they carefully consider the amount of data their application is returning to the client, and keep track of where the SQL Server indexes are placed and how the SQL Server data is arranged. SQL Profiler, the Index Tuning Wizard, and graphical execution plans are very helpful tools for pinpointing and fixing these problem queries.

When using cursor logic, choose the cursor that is appropriate for the type of processing you intend to do. Different types of cursors come with varying costs. You should understand what types of operations you intend to perform (read-only, forward processing only, and so forth) and then choose your cursor type accordingly.

Look for opportunities to reduce the size of the resultset being returned by eliminating columns in the select list that do not need to be returned, or by returning only the required rows. This helps reduce I/O and CPU consumption.

Using Multiple Statements

You can reduce the size of your resultset and avoid unnecessary network communications between the client and your database server by performing the processing on the database. To perform processes that cannot be done using a single Transact-SQL statement, SQL Server allows you to group Transact-SQL statements together in the following ways.

Grouping method	Description
Batches	A batch is a group of one or more Transact-SQL statements sent from an application to the server as one unit. SQL Server executes each batch as a single executable unit.
Stored procedures	A stored procedure is a group of Transact-SQL statements that has been predefined and precompiled on the server. The stored procedure can accept parameters, and can return result sets, return codes, and output parameters to the calling application.
Triggers	A trigger is a special type of stored procedure. It is not called directly by applications. It is instead executed whenever a user performs a specified modification (INSERT, UPDATE, or DELETE) to a table.
Scripts	A script is a series of Transact-SQL statements stored in a file. The file can be used

	as input to the **osql** utility or SQL Query Analyzer. The utilities then execute the Transact-SQL statements stored in the file.

The following SQL Server features allow you control the use of multiple Transact-SQL statements at a time.

Feature	Description
Control-of-flow statements	Allow you to include conditional logic. For example, if the country is Canada, perform one series of Transact-SQL statements. If the country is U.K., do a different series of Transact-SQL statements.
Variables	Allow you to store data for use as input in a later Transact-SQL statement. For example, you can code a query that needs different data values specified in the WHERE clause each time the query is executed. You can write the query to use variables in the WHERE clause, and code logic to fill the variables with the proper data. The parameters of stored procedures are a special class of variables.
Error handling	Lets you customize the way SQL Server responds to problems. You can specify appropriate actions to take when errors occur, or display customized error messages that are more informative to a user than a generic SQL Server error.

Reusing Execution Plans

Performance gains can be realized when SQL Server is able to leverage an existing execution plan from a prior query. There are a number of things the developer can do to encourage SQL Server to reuse execution plans. Transact-SQL statements should be written according to the following guidelines.

- Use fully qualified names of objects, such as tables and views.

  For example, do not code this SELECT:

  ```
  SELECT * FROM Shippers WHERE ShipperID = 3
  ```

  Instead, using ODBC as an example, use the **SQLBindParameter** ODBC function:

  ```
  SELECT * FROM Northwind.dbo.Shippers WHERE ShipperID = 3
  ```

- Use parameterized queries, and supply the parameter values instead of specifying stored procedure parameter values or the values in search condition predicates directly. Use either the parameter substitution in **sp_executesql** or the parameter binding of the ADO, OLE DB, ODBC, and DB-Library APIs.

  For example, do not code this SELECT:

  ```
  SELECT * FROM Northwind.dbo.Shippers WHERE ShipperID = 3
  ```

  Instead, using ODBC as an example, use the **SQLBindParameter** ODBC function to bind the parameter marker (?) to a program variable and code the SELECT statement as:

  ```
  SELECT * FROM Northwind.dbo.Shippers WHERE ShipperID = ?
  ```

- In a Transact-SQL script, stored procedure, or trigger, use **sp_executesql** to execute the SELECT statement:

  ```
  DECLARE @IntVariable INT
  DECLARE @SQLString NVARCHAR(500)
  DECLARE @ParmDefinition NVARCHAR(500)
  /* Build the SQL string. */
  SET @SQLString =
  N'SELECT * FROM Northwind.dbo.Shippers WHERE ShipperID = @ShipID'
  /* Specify the parameter format once. */
  SET @ParmDefinition = N'@ShipID int'
  /* Execute the string. */
  SET @IntVariable = 3
  EXECUTE sp_executesql @SQLString, @ParmDefinition,
  @ShipID = @IntVariable
  ```

sp_executesql is a good alternative when you do not want the overhead of creating and maintaining a separate stored procedures.

Reusing Execution Plans for Batches

If multiple concurrent applications will execute the same batch with a known set of parameters, implement the batch as a stored procedure that will be called by the applications.

When an ADO, OLE DB, or ODBC application will be executing the same batch multiple times, use the PREPARE/EXECUTE model of executing the batch. Use parameter markers bound to program variables to supply all needed input values, such as the expressions used in an UPDATE VALUES clause or in the predicates in a search condition.

Maintaining Statistics on Columns

SQL Server allows statistical information regarding the distribution of values in a column to be created even if the column is not part of an index. This statistical information can be used by the query processor to determine the optimal strategy for evaluating a query. When you create an index, SQL Server automatically stores statistical information regarding the distribution of values in the indexed column(s). In addition to indexed columns, if the AUTO_CREATE_STATISTICS database option is set to ON (which it is by default), SQL Server automatically creates statistics for columns that get used in a predicate even if the columns are not in indexes.

As the data in a column changes, index and column statistics can become outdated and cause the query optimizer to make less-than-optimal decisions about how to process a query. Periodically, SQL Server automatically updates this statistical information as the data in a table changes. The sampling is random across data pages, and taken from the table or the smallest nonclustered index on the columns needed by the statistics. After a data page has been read from disk, all the rows on the data page are used to update the statistical information. The frequency at which the statistical information is updated is determined by the volume of data in the column or index and the amount of changing data.

For example, the statistics for a table containing 10,000 rows may need to be updated after 1,000 index values have changed because 1,000 values may represent a significant percentage of the table. However, for a table containing 10 million index entries, 1,000 changing index values is less significant, and so the statistics may not be automatically updated. SQL Server, however, always ensures that a minimum number of rows are sampled; tables that are smaller than 8 MB are always fully scanned to gather statistics.

Note Outdated or missing statistics are indicated as warnings (table name in red text) when the execution plan of a query is graphically displayed using SQL Query Analyzer. Additionally, monitoring the **Missing Column Statistics** event class using SQL Profiler indicates when statistics are missing.

Statistics can easily be created on all eligible columns in all user tables in the current database in a single statement by using the **sp_createstats** system stored procedure. Columns not eligible for statistics include nondeterministic or nonprecise computed columns, or columns of **image**, **text**, and **ntext** data types.

Creating statistics manually allows you to create statistics that contain multiple column densities (average number of duplicates for the combination of columns). For example, a query contains the following clause:

```
WHERE a = 7 and b = 9
```

Creating manual statistics on both columns together (**a**, **b**) can allow SQL Server to make a better estimate for the query because the statistics also contain the average number of distinct values for the combination of columns **a** and **b**. This allows SQL Server to make use of the index (preferably clustered in this case), if it is built on col1 rather than needing to resort to a table scan. For information on how to create column statistics, see the topic "CREATE STATISTICS" in SQL Server Books Online.

Finding More Information

- SQL Server Books Online provides information on SQL Server architecture and database tuning along with complete documentation on command syntax and administration. SQL Server Books Online can be installed from the SQL Server installation media on any SQL Server client or server computer.
- For the latest information on Microsoft SQL Server, including technical papers on SQL Server, visit the public Microsoft SQL Server Web sites at:

  http://www.microsoft.com/sql

  http://www.microsoft.com/technet/prodtechnol/sql

  http://msdn.microsoft.com/sqlserver
- An external resource that provides good information in the form of a periodical can be found at http://www.sqlmag.com . You will find many optimization and tuning hints, code samples, and insightful articles outlining the internal workings of SQL Server and other valuable information.
- Delaney, Kalen & Soukup, Ron. Inside Microsoft SQL Server 2000, Microsoft Press, 2001.

  This book updates the previous version (Inside Microsoft SQL Server 7.0) with information for SQL Server 2000. This book delves into many of the internal concepts of SQL Server that cannot easily be found elsewhere.
- Kimball, Ralph. The Data Warehouse Lifecycle Toolkit, John Wiley and Sons, 1998.

  This is considered by many to be one of the best data warehousing how-to books written on the subject. It provides excellent insight into data warehouse database design and does a great job of explaining dimensional modeling concepts.
- Celko, Joe. *SQL for Smarties*. Morgan Kaufmann, 1999.

  There is some very helpful information in this book. Contains solutions to common problems such as representing and querying hierarchical data. Chapter 28 is dedicated to optimizing SQL queries.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be

interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This white paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AS TO THE INFORMATION IN THIS DOCUMENT.
