
# SQL Server 2005/2008:
# Performance Tuning & Optimization

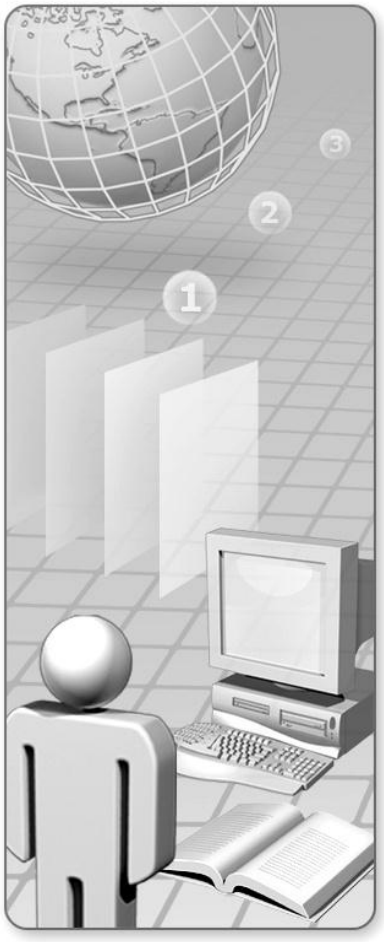
*Student Workbook*

Version 1.3

# Table of Contents

Microsoft | Services

# Incoming Assessment



## Incoming Assessment

This Workshop*PLUS* course includes two 30-minute quizzes – an Incoming Assessment (at the start of the workshop) and an Outgoing Assessment (on the last day of the workshop).

So, you're probably thinking: "Why are they giving me a test during the first hour of this workshop? That is not a very nice way to start the workshop."

We do this because the Assessments provide key data:

- The Incoming Assessment baselines knowledge.
- The Outgoing Assessment measures knowledge transfer.

So, you might be asking yourself: "Key data for whom? What's in it for me? What's in it for Microsoft?" Well, there are benefits for you, benefits for your management, and benefits for the Workshop*PLUS* program. Read on…

## Benefits to you:

- You get an opportunity to see how much you've learned – a measure of improvement.
- Students are not always aware of how much they've learned.
- Students are happily surprised – even amazed – at how much they learn and how much their scores improve.
- You finish the workshop feeling really good because:
  - You know that your hard work was worth it.
  - You feel more confident than ever in your ability to perform well on the job.

The subject matter experts who created this assessment believe that it covers the key points that all students should learn from this workshop. On the last day, after the Outgoing Assessment, the Trainer will review each question and answer, making sure that you understand all the key concepts.

**Note**: Your results are anonymous. Your Assessment results are associated with a Personal Identifier you'll create for yourself or with your student number. In all cases your real name is never associated with your results (More on anonymity below).

## Benefits to your management:

They see positive results that make them confident that the training was worthwhile.

The positive results help to justify the costs of training and perhaps make it possible for them to ask for an increase in training budget.

## Benefits to the Workshop*PLUS* program:

- We obtain data about the quality and value of the workshop.
- We analyze the results to see whether there are problems with:
  - The wording of the questions or the multiple-choice answers.
  - The content in the manual and the labs.
  - The Trainer's knowledge and teaching ability.

## Privacy / Anonymity

Perhaps you're asking yourself: "Who's going to see my results?" There is no need to worry because:

> **Note**: Your name is never recorded anywhere in the Assessment data.

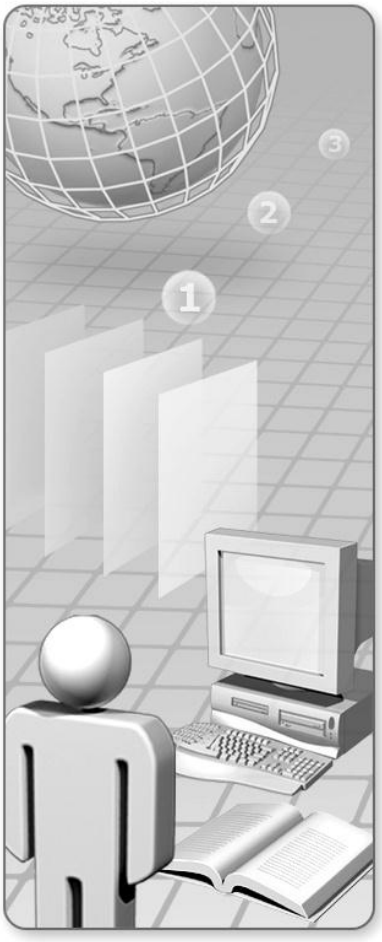
Sometime after the workshop, the scores from the class will be entered into a database. The only data entered is similar to "Student 1 answered question 3 as C." Your real name is never associated with your student number.

No one will see or have access to your individual Assessment scores – not your manager, not others in your company, and not any Microsoft-employed Technical Account Managers, Engagement Managers, or Support Professionals.

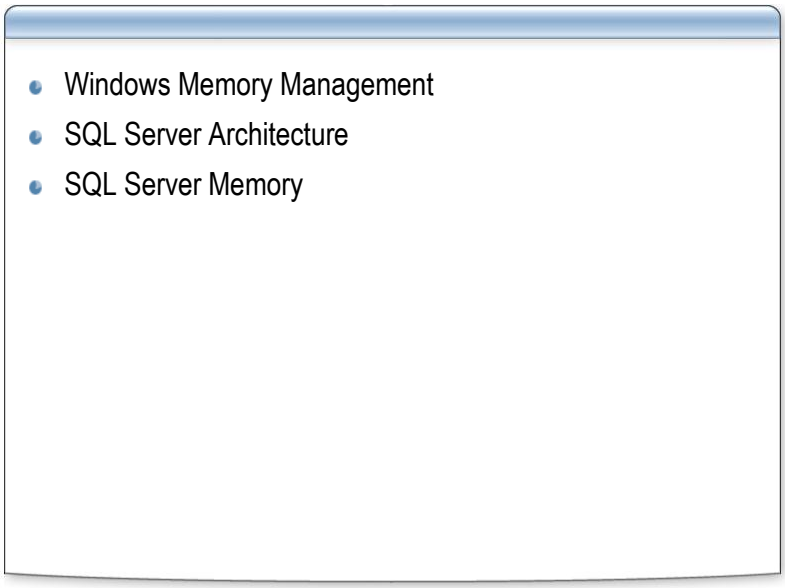
Only aggregate class-average results might be shared with your management.

Microsoft | Services

# Module 1: Architecture



# Module Overview




## Introduction

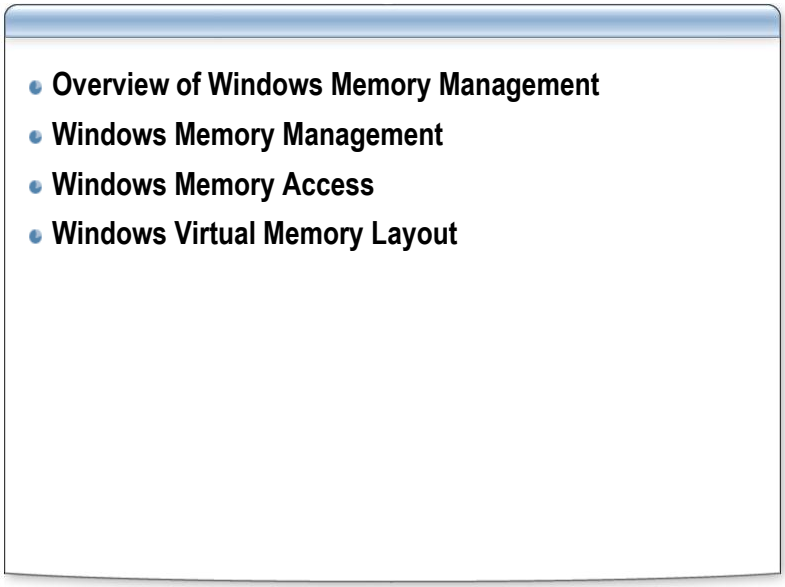
This module provides a brief overview of the server environment in which Microsoft SQL Server 2005 runs. It starts with a high-level overview of Microsoft Windows memory management. Then, it describes the components of the SQL Server architecture and how these components use server resources, such as CPU and memory.

## Objectives

After completing this module, you will be able to:

- Explain the Windows memory management model.
- Describe how the SQL Server Operating System works.
- Explain the SQL Server memory architecture and how to monitor it.

# Section 1: Windows Memory Management




## Introduction

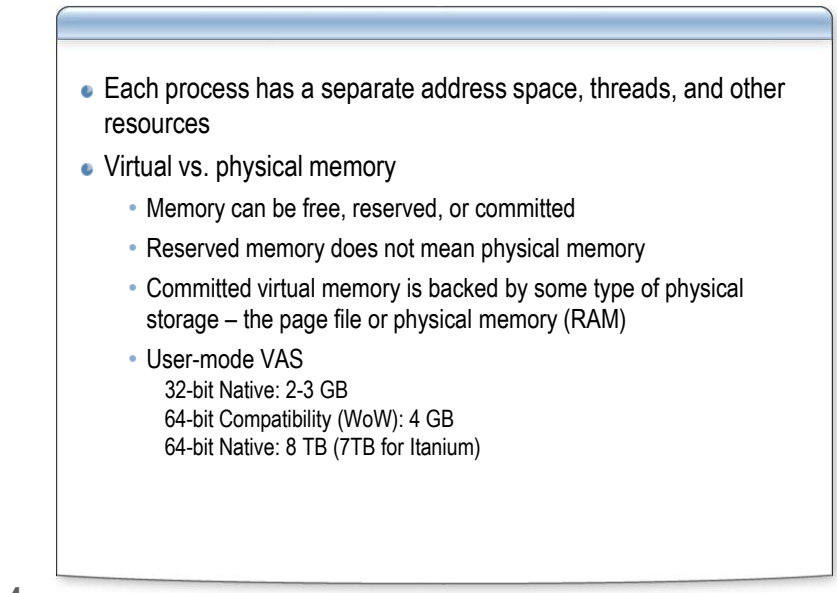
This section provides a brief overview of Windows Server memory management. It explains the concept of virtual memory, the memory capacities of the various processor architectures, how to configure Windows and SQL Server to utilize the available memory, and how memory is accessed.

## Objectives

After completing this section, you will be able to:

- Explain the role of virtual address space (VAS) in protecting Windows applications.
- Explain the process by which Windows switches data between physical memory and the paging file (Pagefile.sys).
- Explain how the Windows memory manager allocates and tracks memory assignments for both the kernel and applications.
- Describe how Windows allocates virtual memory between user mode and kernel mode.

# Overview of Windows Memory Management




The Windows operating system employs various memory-protecting mechanisms to enhance its stability. One such mechanism is the ability to assign a separate, private address space to each process. This space is protected from access by the threads of any other process. One exception to this rule is shared memory where the shared pages of memory are visible to more than one process.

Memory pages within a process can be in one of the following three states:

- **Free**. Free pages are inaccessible.
- **Reserved**. Reserved pages are marked as the territory of the process, but are also inaccessible because there is no committed storage behind the memory to contain the data. Attempts to access reserved memory pages result in an access violation.
- **Committed**. Committed pages are pages for which the Memory Manager has corresponding storage. Committed pages are either located in physical memory or in a disk-based backing store, more commonly known as the page file. Accessing committed memory is permitted and leads to the transfer of the page being accessed to physical memory, if it is not there already.

Windows permits memory to be allocated in a two-step approach—applications can first reserve address space and then commit storage in that address space. It is also possible to both reserve address space and commit storage in one step. When committed memory becomes uncommitted, it becomes reserved memory. Reserved memory becomes free memory when it is released.

Each application that runs under the Windows operating system is composed of one or more processes, with each process represented by a standalone .exe. For example, SQL Server is a service that runs under the sqlservr.exe process. Windows provides protection to application processes by assigning private virtual memory space to each one. This mechanism guarantees that one process does not have direct access to the memory of another process.

## Virtual Address Space

Another term for private virtual memory is Virtual Address Space (VAS). Each process has its own VAS, as well as threads that also cannot be shared with another process. This guarantees that each process has its own resources and does not need to depend on other processes for its resources. The Windows operating system manages the processes and their resources.

> **Note:** For more information about Virtual Address Space (VAS), refer to *Slava Ok's blog* at http://blogs.msdn.com/slavao/archive/2005/01/27/361678.aspx.

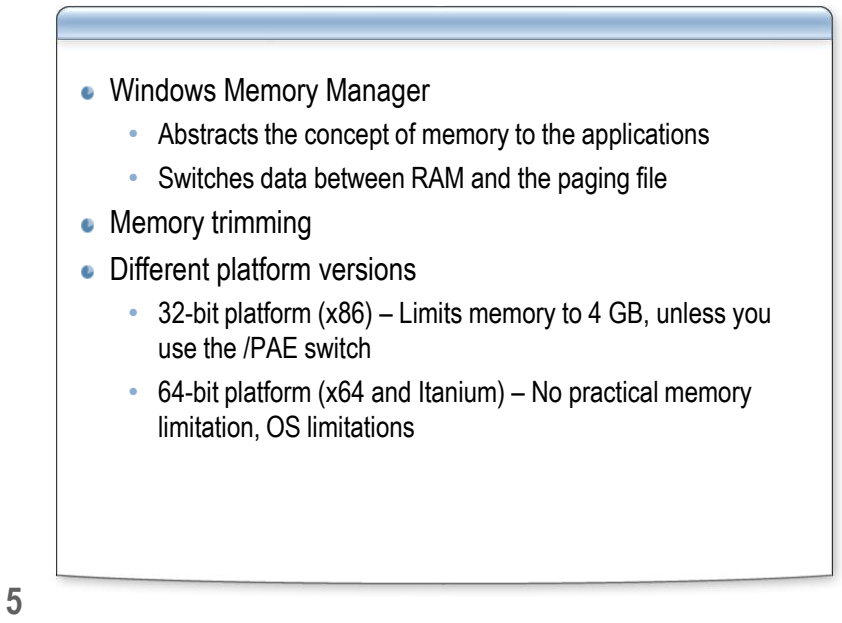
## VAS platform limitations

The available VAS size depends on the platform. The following table describes the VAS limitations applicable to each platform:

| Version | Limitations |
|---|---|
| 32-bit applications running on Windows 32-bit | • VAS is limited to 4 gigabytes (GB) total and is split into two regions—kernel mode and user mode.<br>• The kernel-mode address space contains the critical functionalities of the core operating system (for example, communication with the hardware devices), and it occupies 2 GB.<br>• The user-mode address space ranges in memory from zero to 2 GB. It contains the application executable file and dynamic link libraries (DLLs). You can increase the size of the user-mode address space by using the /3GB flag (Windows 2000 or later) or the /3GB flag in conjunction with the /USERVA flag (Windows 2003 or later). Refer to http://support.microsoft.com/kb/316739 for more information on these two configuration parameters and their effect on memory.<br>**Note:** VAS is always limited to 4 GB for a 32-bit application. Therefore, increasing the user-mode address space shrinks the Kernel Mode address space. A common mistake is to think that this behavior can be changed by using the **/PAE** switch. The **/PAE** switch keeps the overall VAS size untouched, and it only affects the way the Kernel sees the physical memory. |
| 32-bit applications running on Windows 64-bit | • It is possible to run 32-bit applications on a 64-bit computer. This is referred to as running applications in WoW (Windows on Windows) mode. The kernel runs on 64-bit, so it does not need to occupy the same VAS as the 32-bit application. This means that the entire 4 GB VAS is available as user-mode address space, and there is no need to use the /3GB flag.<br>**Note:** There is no way to extend VAS beyond 4 GB for a 32-bit application running in WOW mode. |

| Version | Limitations |
| --- | --- |
| 64-bit applications running on Windows 64-bit | • The process user-mode has 8 terabytes of VAS for x64 and 7 terabytes for IA64. The kernel-mode also has 8 terabytes on 64-bit versions of Windows. |

# Overview of Windows Memory Management (continued)




## Abstracting the concept of memory to applications

The paging file (Pagefile.sys) is a hidden file on the computer hard disk that is used by Windows as if it were random access memory (RAM), also called physical memory. Windows uses both types of storage—the paging file and the physical memory—when maintaining application data while the system is operational. The paging file and physical memory together make up virtual memory, which abstracts the ***memory concept*** and provides generic memory to applications. Applications are unaware of whether they are using physical memory or the paging file.

## Switching data between physical memory and the paging file

One goal of the Windows memory manager is to ensure optimum physical memory usage among the applications because this improves application performance. Windows attempts to give physical memory to most of the applications, whenever possible. However, physical memory is limited, so Windows must page memory to the paging file when physical memory is low. When an application needs to use a memory portion that has been paged to disk, the memory manager loads the data back into memory from the disk.

## Memory trimming

When the computer runs out of physical memory, the Windows memory manager finds physical memory that it has assigned to applications but has not been used. Windows then pages this memory out to disk until it has restored physical memory to acceptable levels. This process is called ***trimming***. Windows cannot be too aggressive in the

trimming process. For example, it is not healthy to keep paging out the memory if only 128 megabytes (MB) are in use and there is still 786 MB left. Windows must balance the memory usage and the trimming process, and it does so by keeping the minimum amount of free physical memory available.

An ideal amount of free physical memory in Windows 2000 ranges from 4 to 10 MB, and when the amount of free memory falls below 4 MB, trimming takes place. However, you should not rely on these numbers because they can change at any time (for example, a Service Pack can modify this behavior).

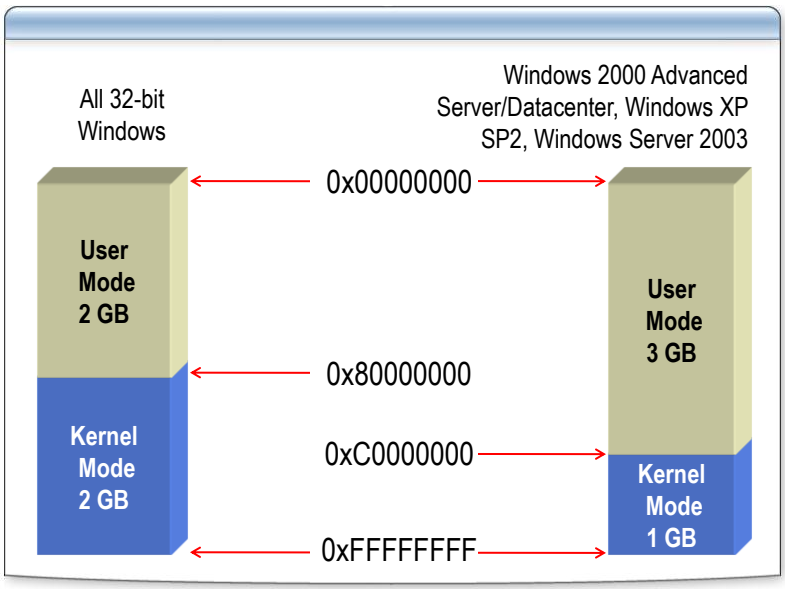
In Windows 2003, the algorithm to determine the ideal amount of free physical memory was adjusted and the range was changed as a result. Windows 2003 provides an application programming interface (API) that enables an application to receive notifications when the computer is running out of memory. This enables applications such as SQL Server to adjust its internal structures, which, in turn, provides the opportunity to return the memory to the operating system. This mechanism helps improve memory management.

## Windows versioning

With the exception of some memory limit restrictions, the characteristics of Windows memory management in different platform versions are very similar. The following table describes these memory limit restrictions:

| Version | Restrictions |
|---|---|
| 32-bit platform (x86) | • By default, the Windows memory manager uses 32-bit pointers to map the memory. This limits the operating system to a maximum of 4 GB of memory. While this is not typically a problem for desktop computers, it is a problem for servers.<br>• You can use the /PAE switch to enable Windows to use larger pointers and, consequently, to address more memory. You set the /PAE switch in the C:\boot.ini file, and you should turn it ON whenever the server has more than 4 GB of physical memory installed. If this switch is not enabled, the 4 GB limit applies even though more memory is installed. |
| 64-bit platform (x64 and Itanium) | • The Windows memory manager uses 64-bit pointers, which do not have any restrictions. In a 64-bit environment, there is no need to use the /PAE flag.<br>• The Windows 2003 Enterprise Edition Service Provider Interface is limited to 1 terabyte. |

# Windows Virtual Memory Layout




Virtual memory is called ***virtual*** because it is a logical view of memory that does not necessarily represent how the physical memory is laid out. The 32-bit Windows operating system uses a virtual memory system based on a 32-bit address space. This amounts to 232 bytes of memory, which is 4,294,967,296 bytes, or 4 GB.

One of the defining parts of a running process is the VAS, which is the set of addresses that the threads of a process are allowed to use. When a process runs, the memory manager maps the VAS to the physical addresses where the data really exists. This mapping is the responsibility of the operating system, and prevents separate processes from bumping into or overwriting the address space of other processes.

Even though the 32-bit Windows systems have access to 4 GB of virtual memory per process, this virtual memory is partitioned between user mode and kernel mode. Based on the research done by the Windows product team, it was discovered that in some cases, Windows does not require a full 2 GB worth of virtual memory for kernel mode. This includes environments where the Windows server hosts a single application such as a single instance of SQL Server. As of Microsoft Windows NT 4.0 Enterprise Edition, Service Pack 3 (SP3), the Windows kernel changed to support a 3 GB user-mode address space, and a 1 GB kernel-mode address space.

You can enable this feature by using the /3GB switch in the boot.ini file. Otherwise, the user-mode/kernel-mode virtual memory equation remains at 2 GB each. Additionally, programs will not use this extra 1 GB of user-mode address space unless they are explicitly set to do so.

On 32-bit Windows systems with 2 GB of user-mode space and 2 GB of kernel-mode space, a memory range of 0x00000000–0x7FFFFFFF is for user mode and 0x80000000–0xFFFFFFFF for kernel mode.

On Windows systems with 3 GB of user-mode space, and 1 GB of kernel-mode space, this implies a range of 0x00000000–0xBFFFFFFF for user mode and 0xC0000000–0xFFFFFFFF for kernel mode.

Windows uses the kernel-mode space in the process for kernel-mode components and user-mode components that may have implications for other applications, such as data transfers, etc. In general, an application cannot address anything in the virtual memory section directly, so it is best to assume that any kernel-mode memory area is inaccessible. If an application attempts to address anything in a virtual memory space, Windows generates an access violation.

## Considerations for using the /3GB switch

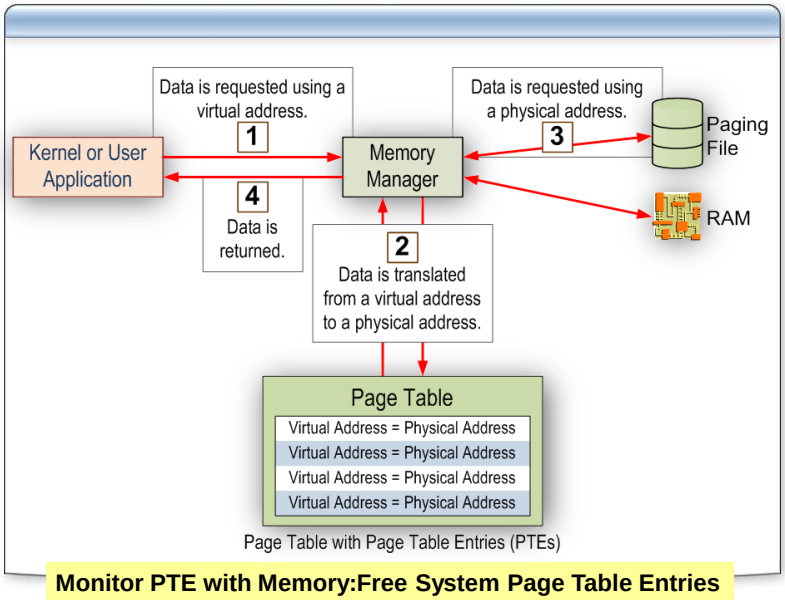
When using the /3GB switch, you must consider the following:

- Processes that use a large number of handles may use up the page pool memory.
- Processes that add users to a large number of security groups will cause the security token for these users to bloat. This may cause the page pool to deplete.
- The kernel uses up the Free System page table entries (PTEs) on heavily loaded servers that are configured with the /3GB switch. This results in server instability such as random network problems (the server drops packets or can no longer be reached), which might require a system restart.

**Note:** Memory management is detailed in the book ***Microsoft Windows Internals*** by Mark E.Russinovich and David A.Solomon, Microsoft Press, 2005.

**For more information on the use of the /3GB and /PAE switches refer to the Knowledge Base article, Q291988: *A description of the 4 GB RAM Tuning feature and the Physical Address Extension parameter*** at http://support.microsoft.com/kb/291988.

# Windows Memory Access



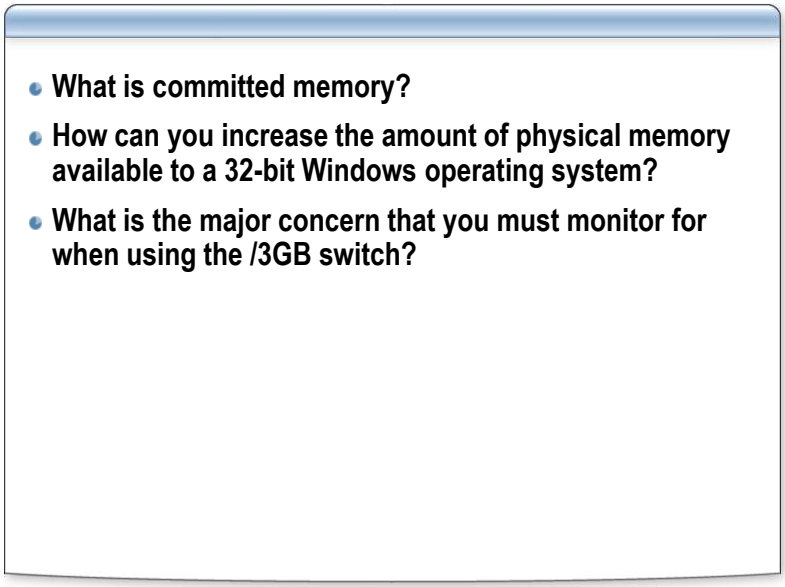

7 **Monitor PTE with Memory:Free System Page Table Entries**

User-mode processes or applications are never able to directly write to real memory and never actually know where their data resides. A user-mode process or application can request a block of memory and write to it. The data written to the memory location might be written either to real memory or to a paging file.

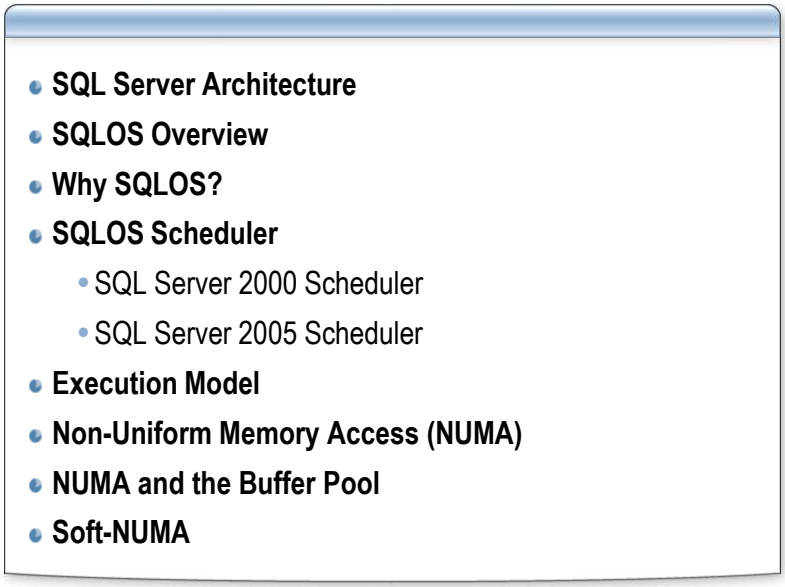
The Windows memory manager is responsible for allocating and tracking memory assignments for both the kernel and applications. The memory manager translates VAS used by the operating system and applications to actual physical memory locations. The translation of virtual memory to physical memory is transparent to the applications, so the memory manager must index the location where the data resides in real memory or in a page file. When an application requests for the data, the memory manager can refer to the index to determine where that data actually resides and then retrieve it for the application.

The Page Table, one of the internal tables used by the memory manager in translating VAS to physical addresses, resides in the memory space of the kernel.

# Section 1 Review



# Section 2: SQL Server Architecture




## Introduction

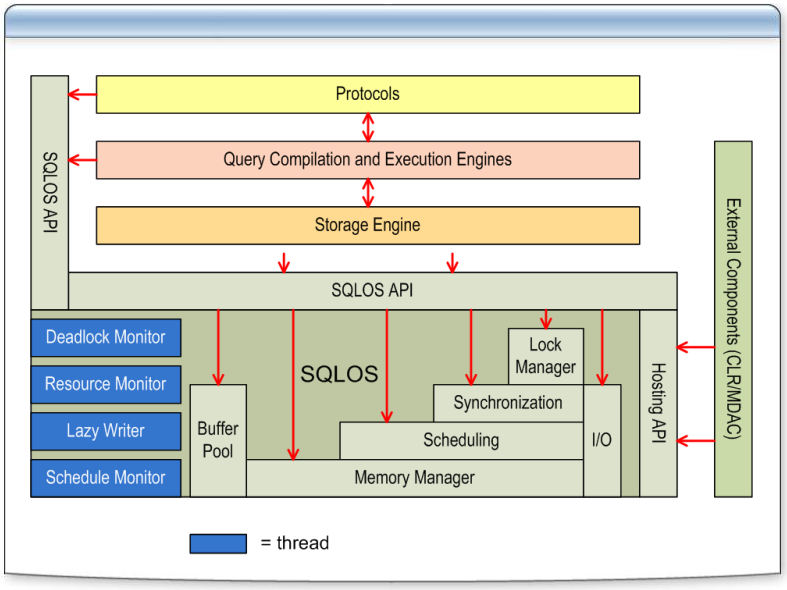
This section introduces the SQL Server architecture, including the SQLOS, the SQLOS scheduler, the execution model, and the non-uniform memory access (NUMA) architecture. A solid understanding of the SQL Server architecture will set the foundation for understanding performance-related areas such as waits, the various execution queues and lists, along with how to interpret data from advanced architectures such as NUMA.

## Objectives

After completing this section, you will be able to:

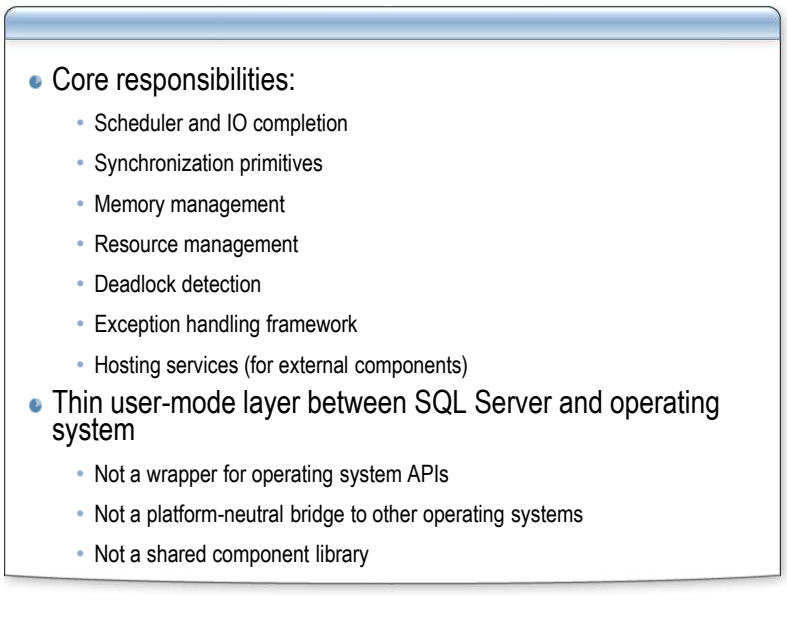
- Identify the main components of the SQL Server architecture.
- Describe the functions of SQLOS.
- Explain the additional benefits SQLOS provides to SQL Server over the more generic Windows operating system.
- Explain the role of the SQLOS scheduler.
- Compare the SQL Server 2005 scheduler with that in SQL Server 2000.
- Differentiate between the three queues in the SQL Server execution model.
- Explain the benefits of NUMA.
- Explain how NUMA affects the buffer pool as compared to non-NUMA architectures.
- Configure SQL Server to use soft-NUMA.

# SQL Server 2005 Architecture




10

# Overview of SQLOS



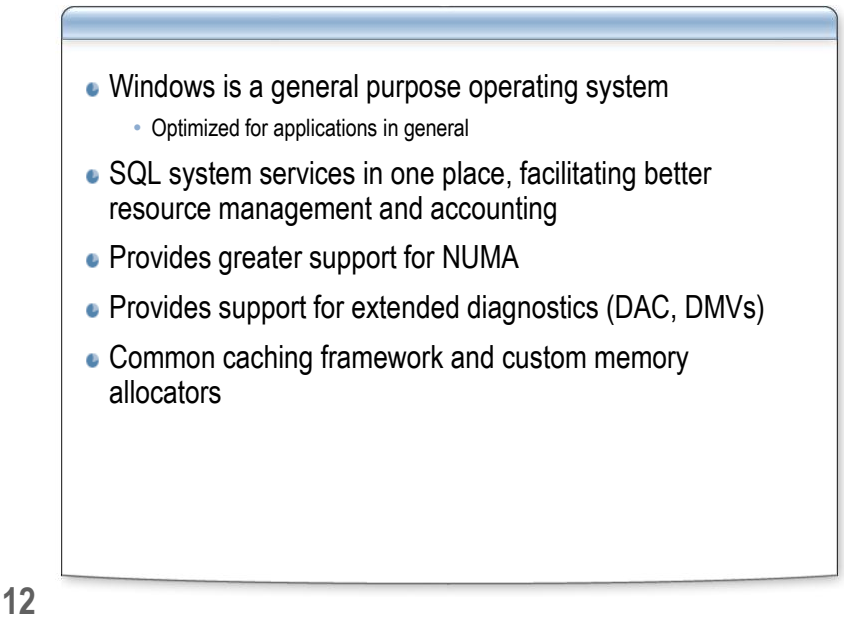

When SQL Server starts, each logical processor is allocated one SQLOS scheduler.

- Scheduler and input/output (I/O) completion
- Synchronization primitives
- Memory management
- Resource management
- Deadlock detection and management
- Exception handling framework
- Hosting services for external components such as common language runtime (CLR)

SQLOS is a very thin user-mode layer. Note that SQLOS is not:

- A wrapper for operating system APIs.
- A bridge to other operating systems.
- A shared component library.

## The Need for SQLOS




Hardware technology and architectures continue to evolve. The introduction of SQLOS allows SQL Server to be able to respond to these new architectures with greater flexibility and control. One area where this is evident is the improved support for the non-uniform memory access (NUMA) architectures of large servers.

Windows is a general purpose operating system that is optimized to work with as many applications as possible, not particularly SQL Server. For SQL Server to derive the maximum benefit of the underlying platform, it must have a finer degree of control over the operating system. SQLOS manages all SQL system services centrally, thereby leading to better accountability and management of resources.

SQLOS allows SQL Server to take advantage of the more advanced hardware architectures that are being introduced in the marketplace. SQLOS uses the same object hierarchy as the NUMA architecture with respect to nodes. This enables SQLOS to take advantage of the scalability enhancements of the NUMA architecture over the symmetric multiprocessor (SMP) architecture.

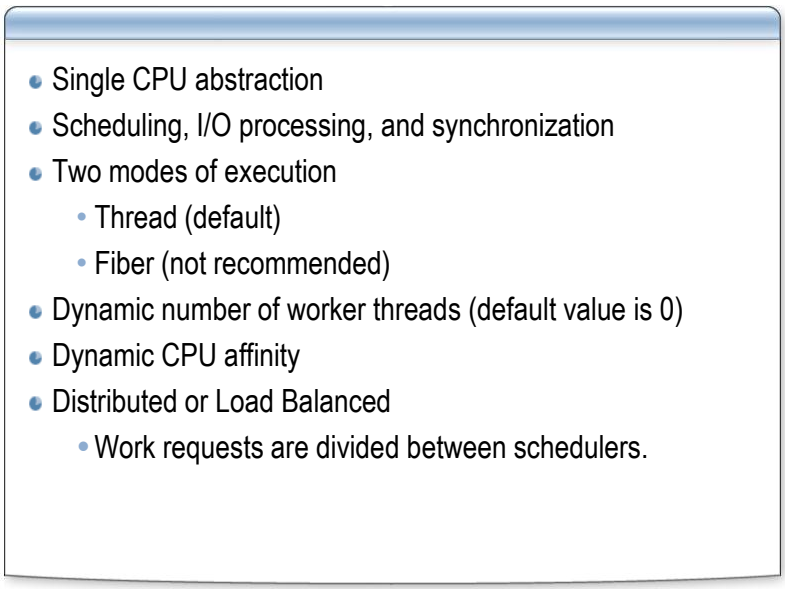
Additionally, SQLOS provides extended diagnostic capabilities, such as the Dedicated Admin Connection (DAC) and dynamic management views (DMVs), thereby facilitating better troubleshooting and debugging of issues.

SQL Server contains various parts that require the use of cache and memory allocation, including areas such as the Optimizer, Query Engine, and Storage Engine. In the previous versions of SQL Server, each of these areas had its own code to handle the caching and memory allocation needs. SQLOS brings all of this disparate code into one central

location, thereby resulting in a common process for managing cache needs across all areas of SQL Server. In addition, the process by which memory is allocated is now handled by custom memory allocators, depending on the memory allocation requirements. Now, there are page allocators, virtual allocators, and shared memory allocators.

**Note**: For more information on SQLOS, refer to the blog ***SQLOS's memory manager and SQL Server's Buffer Pool*** at http://blogs.msdn.com/slavao/archive/2005/02/11/371063.aspx.

## The SQLOS Scheduler




When SQL Server starts, each physical processor is allocated one SQLOS scheduler.

The SQLOS scheduler helps schedule work tasks by using *non-preemptive scheduling*. This means that a task is usually allowed to run until its quantum has been reached or until it suspends (it needs to wait) on some synchronization event. This quantum is the one that is determined within the SQLOS code, not the operating systems. This helps reduce expensive CPU context switches.

The SQLOS schedulers also provide support for non-preemptive IOs, which are asynchronous disk IOs that complete during a scheduler context switch (therefore the asynchronous nature). The scheduler helps with synchronization in a manner that if a task needs to wait for a resource, the thread running the task is removed from the scheduler and placed in a waiting list until the resource becomes available. This helps ensure that other tasks can get the work done rather than being blocked behind the one waiting for a resource, such as a lock or disk IO to complete.

A task in SQLOS is a unit of execution that is associated with either a thread or a fiber. It is the thread that is assigned to the CPU for execution. A fiber is a lightweight execution context that also helps reduce the cost of context switching. Due to various performance issues seen with the use of fiber mode, we recommend using the thread mode unless extensive, thorough testing shows that fiber mode will meet performance service-level agreements (SLAs) without any undue side effects.

In SQL Server 2005, there is a critical change in the scheduling configuration from previous versions. In SQL Server 2000, the default value of the *max worker threads*

option is 255. In SQL Server 2005, the default value of this option is 0, which causes the scheduler to generate a dynamic number of worker threads based on the following:

- For x86 systems with four or less CPUs, SQL Server starts with 256 threads assigned evenly across the schedulers. For every additional CPU over four, SQL Server adds eight extra threads. The formula for calculating the value of the **max worker threads** option is:

```
Maximum worker threads = (256+ (<processors> -4) * 8)
```

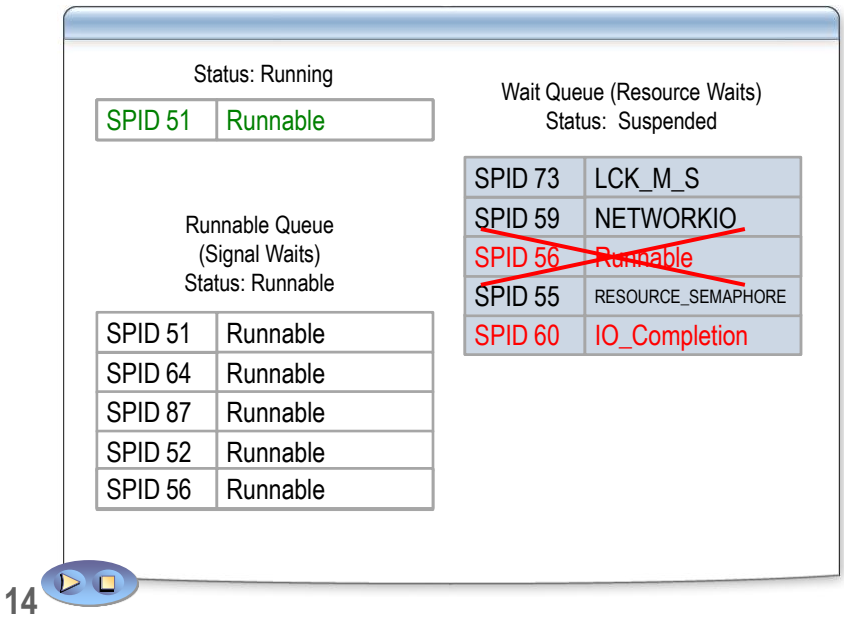
- For 64-bit systems with four or less CPUs, SQL Server starts with 512 threads. For every additional CPU over four, SQL Server adds 16 extra threads. The formula for calculating the max worker threads option is:

```
Maximum worker threads = (512+ (<processors> -4) * 16)
```

Another improvement that SQLOS brings to SQL Server 2005 is the **CPU affinity**. The SQL Server 2005 CPU affinity is dynamic, and you can configure it at any time, without recycling the SQL Server service. This not only improves the availability of the SQL Server service but also helps respond to resource requirements in a more controlled fashion.

SQLOS distributes or load balances work requests between schedulers based on the number of active tasks assigned to each scheduler. This helps ensure that the work is balanced across the various schedulers and prevents a common issue seen in SQL Server 2000 where certain schedulers would be overloaded while others are idle.

# Execution Model




A good understanding of the execution model of SQL Server 2005 helps a Database Administrator (DBA) troubleshoot performance problems much more efficiently. This topic covers the various states of a server process identifier (SPID), the lists and queues involved, and how a DBA can determine certain resource bottlenecks issues from this information.

The slide above shows the SQL Server 2005 execution model on one of the processors on a computer. At any given time, on a single scheduler, you can have one running SPID. On a four-processor computer, there could be up to four running SPIDs at any given time.

> **Note**: You can determine the SPID status by using the **sys.dm_exec_requests** and/or **sys.dm_exec_sessions** DMVs. SQL Server Books Online documents the Status column of these two DMVs.

SQL Server 2005 maintains the following three queues:

- **Runnable queue**. The queue contains SPIDs that are waiting for their turn to run on the CPU. The wait for CPU time is referred to as ***signal wait time***. You can detect a CPU resource bottleneck by checking if the signal wait time from the sys.dm_os_wait_stats DMV indicates greater than 25 percent of the total wait time. The Runnable status indicates that the request (SPID) is running, but is temporarily scheduled out because it has exceeded its quorum on the CPU. It also indicates that a SPID that has finished waiting for a resource and is now ready for its quorum on the CPU.

- **Wait queue**. This queue holds those SQL Server SPIDs that are waiting for specific resources that are required to continue running. In the example shown on the slide above, SPID 73 is waiting for a lock, SPID 59 is waiting for NETWORKIO, SPID 56 is waiting for a parallel thread to complete (CXPACKET), and SPID 55 may be waiting for a memory grant (RESOURCE_SEMAPHORE) before it can run. Note that the status of these SPIDs will show as ***Suspended*** because they are waiting for resources or some event to complete before they can proceed.

  The sys.dm_os_wait_stats DMV is an important tool that helps monitor the various waits experienced by SPIDs. A comparison of the results of two queries taken at different times from this DMV helps identify the resources that are being waited on the most. This can help with capacity management and planning as well as help isolate performance issues.

  The sys.dm_os_waiting_tasks DMV presents more detail about the SPIDs in the Wait queue. This DMV provides specific wait resources; wait durations for the specific SPID, as well as useful information to help isolate blocking problems due to resource waits.

- **Running queue**. This queue refers to the SPID that is currently running.

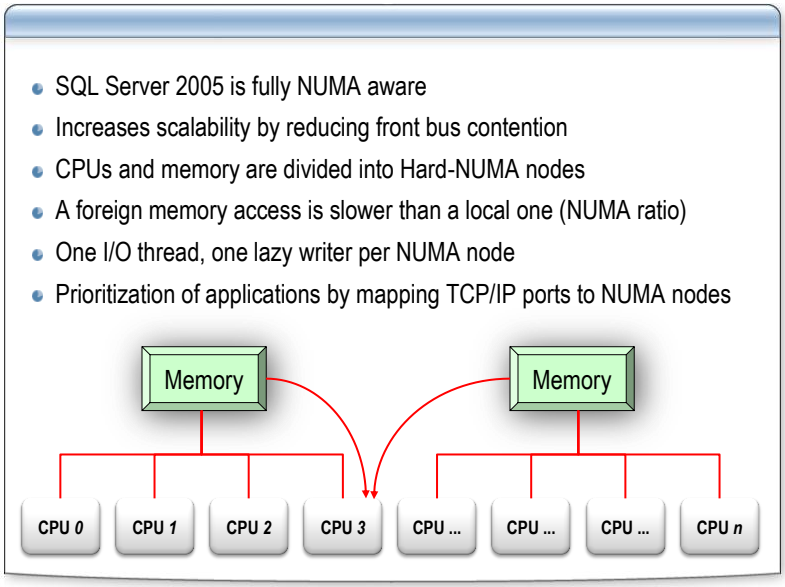
## The Execution model flow

When an SPID is running, it might have an immediate need for I/O, so it moves from the running list to the waiter list, and the next available Runnable SPID moves to the ***running*** status. When any of the SPIDs in the waiter list gets the resource that it was waiting for (CXPACKET in the example shown on the slide above), it moves to the runnable queue. This signifies that the resource is now available. By the time the SPID finishes execution, it may cycle through this process multiple times.

This flow illustrates how SQL Server 2005 governs execution when SPIDs are waiting for resources. You can use the **sys.dm_os_wait_stats** DMV to get information on the waits encountered by threads that are in execution.

> **Note:** For more information about SQL Server wait types, refer to the topic *Description of the waittype and lastwaittype columns in the master.dbo.sysprocesses table in SQL Server 2000 and SQL Server 2005* at http://support.microsoft.com/kb/822101.
>
> For more information about the different wait types, refer to the topic *sys.dm_os_wait_stats* in SQL Server 2005 Books Online.

# Non-Uniform Memory Access (NUMA)




One of the major advantages of SQLOS is the flexibility that it offers in adjusting SQL Server to new and advanced hardware architectures without impacting the other areas of SQL Server code. With SQL Server 2005 SQLOS, SQL Server is now fully NUMA aware.

One major problem with the SMP architecture in high CPU count systems is the contention on the system bus, sometimes referred to as the ***memory front*** or ***front-side bus***. This is because with SMP, all CPUs use the same bus to access memory. The more number of CPUs use the bus, the more is the contention. Increasing the speed of the system bus has only somewhat alleviated the problem. NUMA was designed to bypass the single bus architecture.

NUMA architecture groups CPU and Memory into nodes. Nodes include their own CPUs and memory, and in some cases, their own I/O channels. Each node has its own system bus for the CPUs to reference the node memory. There are interconnects between the nodes to allow the memory to be referenced by other nodes.

Memory local to a node is called local memory. Memory located in another node is referred as foreign or remote memory. Access to foreign memory is often more expensive in time than accessing local memory. The cost of local memory access over the cost to foreign memory access is called the **NUMA ratio**. A higher NUMA ratio implies a higher performance impact.

SQL Server 2005 also allows non-NUMA hardware to be grouped into NUMA nodes through the soft-NUMA option. This allows SQL Server 2005 to take advantage of the NUMA architecture even if it is not running on the NUMA hardware.

> **Note**: For a complete reference on how to configure soft-NUMA, refer to SQL Server Books Online.

SQL Server takes advantage of the NUMA architecture by allocating an I/O thread and a lazywriter thread per NUMA node. This helps reduce any possible performance bottlenecks with these two normally single thread tasks.

Another area where SQL Server improves resource management is the ability to allow the mapping of specific TCP/IP ports to specific NUMA nodes. Applications can then be directed to connect to the relevant TCP/IP port and have predictable performance, given the resources available on the NUMA node. This feature can help the high priority applications ensure that they get the necessary resources that they require.

To determine if SQL Server is running on a NUMA platform, you can review the SQL Server errorlog. The following errorlog entries would indicate a two-node NUMA configuration:

```
Multinode configuration: node 0: CPU mask: 0x0000000a Active CPU mask: 0x0000000a.
This message provides a description of the NUMA configuration for this computer.
This is an informational message only. No user action is required.

Multinode configuration: node 1: CPU mask: 0x00000005 Active CPU mask: 0x00000005.
This message provides a description of the NUMA configuration for this computer.
This is an informational message only. No user action is required.
```

The sys.dm_os_memory_clerks DMV returns results for each NUMA memory node. With SQL Server 2008, you can use the sys.dm_os_memory_nodes and sys.dm_os_nodes DMVs to track NUMA node resources.

With regards to performance monitoring, one area of interest will be the amount of foreign page references that are occurring. You can use System Monitor to track this information. The ***SQL Server:Buffer Node*** object has the foreign page counter; this value should be as low as possible.

## NUMA and the Buffer Pool

When SQL Server starts up, memory allocated to the buffer pool is evenly distributed across the NUMA nodes. However, if the available memory on each node is not equal (which might be the case if other services are running when SQL Server starts and are consuming memory on one node but not others), then the memory allocated to the buffer pool in any given node may be a mix of both local and foreign memory.

If SQL Server is assigned to a subset of the available NUMA nodes, the memory for the buffer pool is not limited to the local memory on the assigned nodes. To prevent foreign memory from being allocated to the buffer pool in this case, you need to properly set the max server memory configuration.

Similarly, if a NUMA node is taken away from a running instance through an affinity mask change, the local memory of that node is redistributed to the other nodes as foreign memory. To prevent this, you should decrease the max memory setting accordingly.

You can use the System Monitor SQL Server:Buffer Node Target Pages and Foreign Pages counters to monitor the distribution of buffer pool memory in NUMA architectures.

Each NUMA node has its own lazywriter thread. This thread is also responsible for the checkpoint operations. Because each NUMA node has its own thread, each node can perform its own checkpoint operations. This helps improve performance and reduces the impact of any bottleneck with checkpoints that might be seen on an SMP system.

When a query scans a table, only the buffer pool on the NUMA node that the query is running on will be used to handle the pages. The only exception is when there is a parallel operation that involves other nodes as well. As a best practice, we recommend you to set the max degree of parallelism setting to the number of CPUs within a single NUMA node.

## Soft-NUMA

The soft-NUMA configuration ability in SQL Server 2005 allows SQL Server to take advantage of NUMA architectures even on hardware that is not built as a NUMA system. Soft-NUMA also enables a DBA to subdivide a NUMA node into smaller groups.

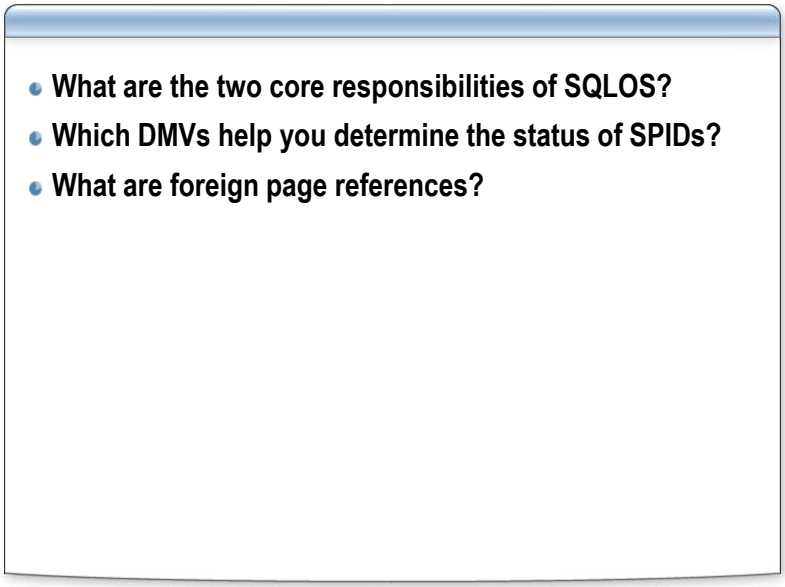
You can configure soft-NUMA by using the registry and affinity mask configuration parameter.

There are a couple of areas that you need to be aware of when configuring soft-NUMA. The first is that memory will be global across all configured soft-NUMA nodes. Therefore, while soft-NUMA allows the grouping of CPUs, it does not allow for the grouping of memory. The exception is when there are underlying hardware NUMA nodes. In this case, only local memory in the hardware NUMA node is global to the groups created by soft-NUMA. The second area of concern is that CPUs from different hardware NUMA nodes cannot be grouped into the same soft-NUMA configured group.

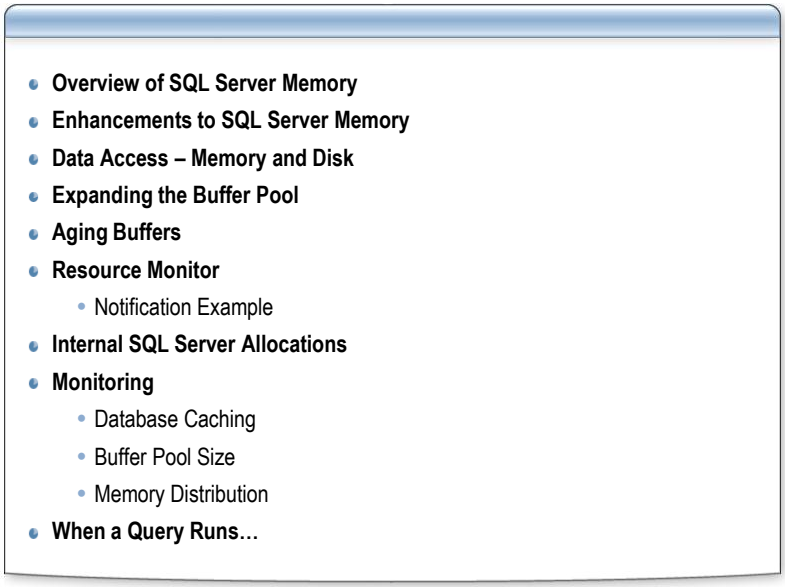
> **Note**: For a complete guide on configuring soft-NUMA, refer to *How to: Configure SQL Server to Use Soft-NUMA* on SQL Server Books Online at http://msdn.microsoft.com/en-us/library/ms345357(SQL.90).aspx.

Soft-NUMA provides the same advantages as hardware NUMA with respect to the I/O and lazywriter threads as well as the ability to map TCP/IP ports to soft-NUMA nodes.

# Section 2 Review



# Section 3: SQL Server Memory




## Introduction

This section describes how SQL Server manages and utilizes memory. It explains how the buffer pool can be increased, how buffer pages can be managed to allow new pages to be brought in from disk, and how SQL Server manages memory in conjunction with the Windows operating system requirements. In addition, the section describes how to monitor memory.

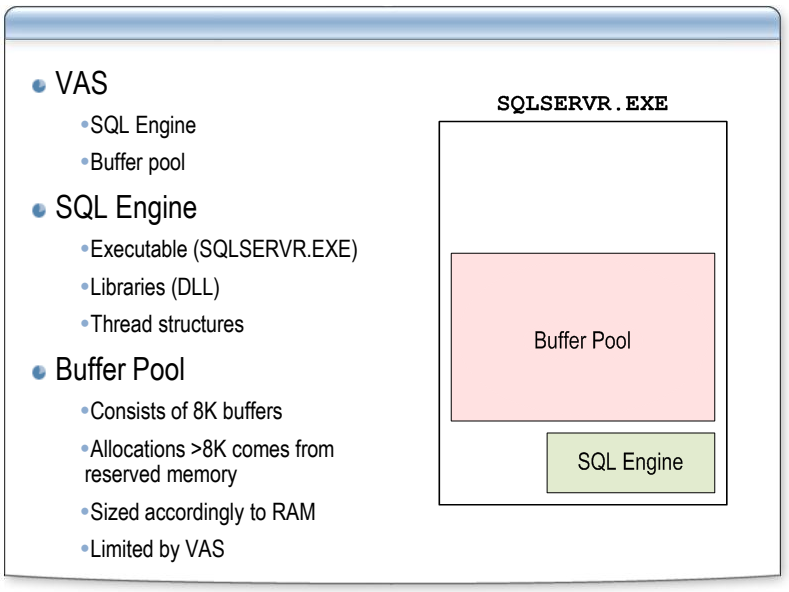
## Objectives

After completing this section, you will be able to:

- Identify the components that reside in a SQL Server process, including SQL Engine and the buffer pool.
- Identify the memory enhancements made in SQL Server 2005.
- Explain the client/server communication process that occurs when SQL Server accesses data.
- Differentiate between the different approaches that you can use to expand the buffer pool, including /3GB and Address Windowing Extensions (AWE).
- Explain how SQL Server manages aging buffers.
- Describe how to troubleshoot memory issues in SQL Server by using Resource Monitor.
- Obtain diagnostic memory information by using the sys.dm_os_ring_buffers DMV.
- Explain how memory clerks monitor the internal SQL Server memory allocations.

- Monitor database caching, buffer pool size, and memory distribution.
- Identify when bottlenecks can occur during query execution.

# Overview of SQL Server Memory




This topic describes the SQL Server virtual address space and the two main components that allocated within it.

## VAS

SQL Server runs as the SQLSERVR.EXE process with its own VAS. During the SQL Server service startup, you can see the SQL Engine and buffer pool inside this VAS space. In the case of a server that has several SQL Server instances, there will typically be one SQLSERVR.EXE process for each of the instances.

## SQL Engine

There is no strict definition for SQL Engine, but for this workshop discussion, it refers to:

- Executable image (SQLSERVR.EXE)
- DLL libraries (NTDLL.DLL, KERNEL32.DLL, ADVAPI32.DLL, USER32.DLL, etc.)
- Thread structures (such as thread stack)
- Other resources needed for SQL Server

## Buffer pool

This is one of the main components of SQL Server. It corresponds to the database cache. Instead of posting an I/O operation to the disk, the data is looked up from the cache in memory. The goal is to use as much memory as possible to have optimum cache performance. However, it is not possible to keep growing the cache indefinitely. The resources that normally limit the buffer pool size are:

- **Physical memory (RAM) available**. The buffer pool keeps growing while the physical memory (RAM) is available (or until the max server memory configuration value is reached). Although the allocation rate can be more aggressive, it can never over allocate memory beyond what is available. If it did, Windows memory manager would start the trimming process, thereby causing catastrophic results to the performance. Instead of paging out memory from the buffer pool to the disk, SQL Server prefers freeing memory by shrinking the cache size.
- **VAS available**. In a 32-bit platform, it is quite common to see a large amount of physical memory that does not fit in the user address space (2 GB). Running out of VAS puts the SQL Server process in a critical situation, and can lead to unexpected errors. For example, it might lead to linked server failures, access violations in extended procedures, errors generating dump files, or slow backup performance. Therefore, the buffer pool must leave some VAS available. The default value is 384 MB, but you can adjust it according to the formula shown below. Note that the information is applicable to 32-bit processors.

```
SQL Server Reserved Memory = MemReserved + (NumThreads * 0.5MB)
```

Where:

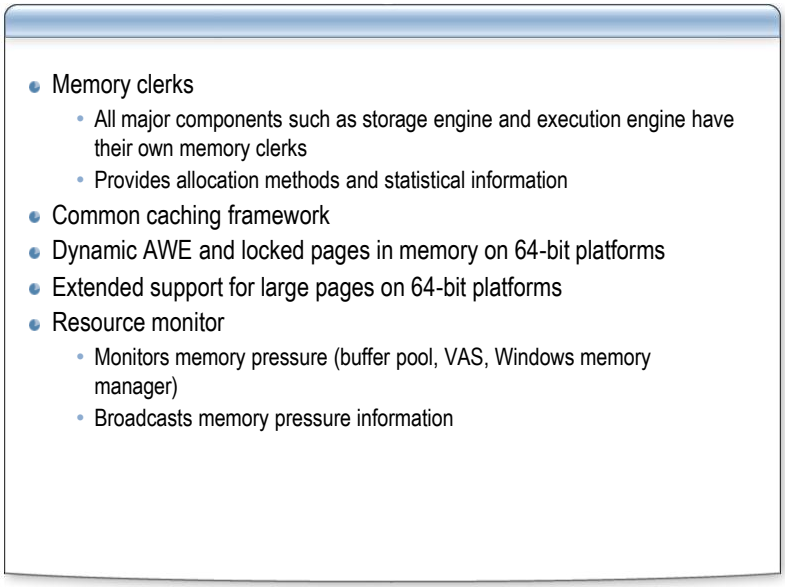
- NumThreads = <the total number of worker threads configured for SQL Server>
- MemReserved = 256 MB, but it can be changed by using the parameter -g during SQL Server startup

The 64-bit processors use different stack sizes. The stack size for X64 processors is 2 MB per stack, whereas the stack size for IA64 processors is 4 MB per thread.

This mechanism enables the buffer pool to reserve huge blocks of memory to store cached data, and it also guarantees that a sufficient amount of VAS will be left (reserved memory) for the respective consumers. In general, VAS is rarely an issue for 32-bit computers with less than 2 GB.

In an x64-bit platforms, Windows-based applications have a user-mode address space of 8 terabytes. IA64 Windows-based applications have 7 terabytes of user-mode address space.

# Enhancements to SQL Server Memory




SQLOS provides access to virtual, physical, shared, and Address Windowing Extensions (AWE) memory.

Each component of SQL Server has its own memory clerk that helps provide memory services required for that component.

SQLOS also provides a common caching framework. This framework allows for caches that meet the requirements of different cached objects. For example, objects with similar cost requirements for cache can share the same cache, but objects with a different caching cost will have a different cache. However, even in the latter case, all caches are managed via the same code base.

The ability to have dynamic thread counts in SQL Server 2005 coupled with the new AWE memory management capabilities in SQLOS enables SQL Server 2005 to respond to memory pressures more efficiently than previous versions. AWE memory is now dynamically managed, rather than statically managed as in SQL Server 2000.

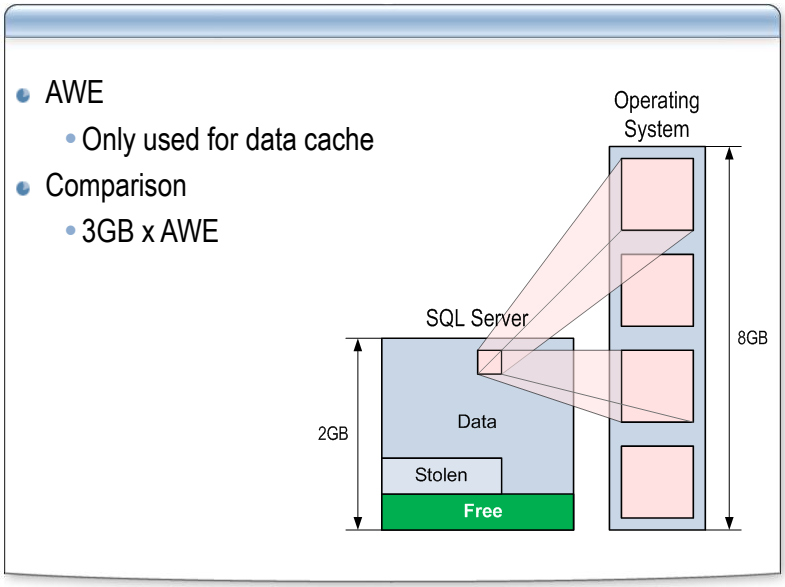
On 64-bit systems, the **lock pages in memory** option is now available. In some workloads, this setting has improved the performance because pages allocated via the AWE mechanism cannot be paged out of memory. 64-bit SQL Server versions will still use the OS's AWE mechanism when the lock pages in memory option is set. SQL Server 2005 64-bit versions also support large page allocations. Large page support allows an application to allocate large page sizes in excess of the native page size of a given processor. Generally, the minimum size of a large page is 2 MB, with the maximums quite a bit larger. IA64 large pages can be 16 MB in size. You must observe precaution

with large page support because the allocations must come from contiguous memory. The longer a server runs, the more fragmented the memory becomes and the successful allocation of large pages becomes more difficult.

The Resource Monitor works in conjunction with the Windows Memory Manager to monitor the availability of virtual and physical memory and to respond to memory pressures through notification mechanisms. These notifications are then passed to memory clerks who attempt to free the memory, if possible, from the various SQL Server components.

SQL Server 2000 did not respond to memory pressures effectively, but SQL Server 2005 can. The latter can truncate the thread pool or the buffer pool even though AWE is enabled. This enables you to dynamically change the AWE memory in response to memory pressure.

# Expanding the Buffer Pool




On 32-bit servers, it is common to exhaust VAS because processes are bound to a 4 GB memory limitation. To address this problem, Windows exposes an API called Address Windowing Extensions (AWE). SQL Server takes advantage of AWE to expand the VAS limit.

> **Note**: A 64-bit environment has a much larger VAS and therefore, does not require AWE.

AWE enables applications to acquire physical memory, and then dynamically map views of this memory to the 32-bit address space. Although using AWE poses additional overhead compared to using memory in 64-bit environments, AWE offers faster data storage than when storing data to disk.

> **Note**: For a complete description of the requirements and how to configure SQL Server to use AWE for memory access, refer to the topic *Enabling Memory Support for Over 4 GB of Physical Memory* at SQL Server Books Online.

## Comparing AWE and /3 GB

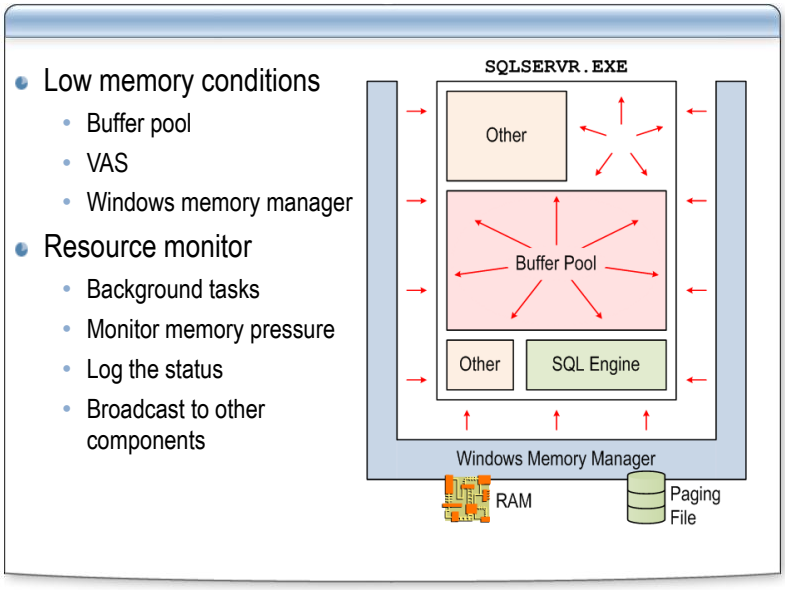
Standard 32-bit addresses can map a maximum of 4 GB of memory. Therefore, the standard address spaces of 32-bit Microsoft Windows Server processes are limited to 4 GB. By default, 2 GB is reserved for the operating system, and 2 GB is made available to the application.

When deciding to use /3GB or AWE, you should consider the following:

- /3GB alone does not enable SQL Server to utilize more than 3 GB of physical memory.
- /3GB increases the user-mode VAS. The procedure cache takes advantage of this setting because it uses stolen pages.
- Enabling /3GB consumes additional kernel resources, such as Page Table Entries (PTEs).
- Computers with more than 16 GB cannot use /3GB because they lack sufficient memory space for the PTE to properly address memory above this amount.
- AWE enables SQL Server to use more than 4 GB of memory.
- The impact of AWE on the number of PTEs is low.
- The procedure cache does not directly benefit from AWE because AWE is used only for data pages.
- Mapping and unmapping pages via AWE poses additional overhead.

You can use both AWE and /3 GB together if you want to use more than 4 GB of memory and also have a larger VAS. However, we highly recommend you to monitor the *Memory: Free System Page Table Entries* counter to ensure that there are sufficient free PTEs. There is no hard threshold for the value of this counter. Only testing and monitoring can establish a valid threshold for any given environment. Processes running on the server other than SQL Server also have a large influence on the value of this counter. If a solution requires both AWE and /3GB for adequate performance, it may indicate the need to transition the SQL Server database server to 64-bit.

# Resource Monitor



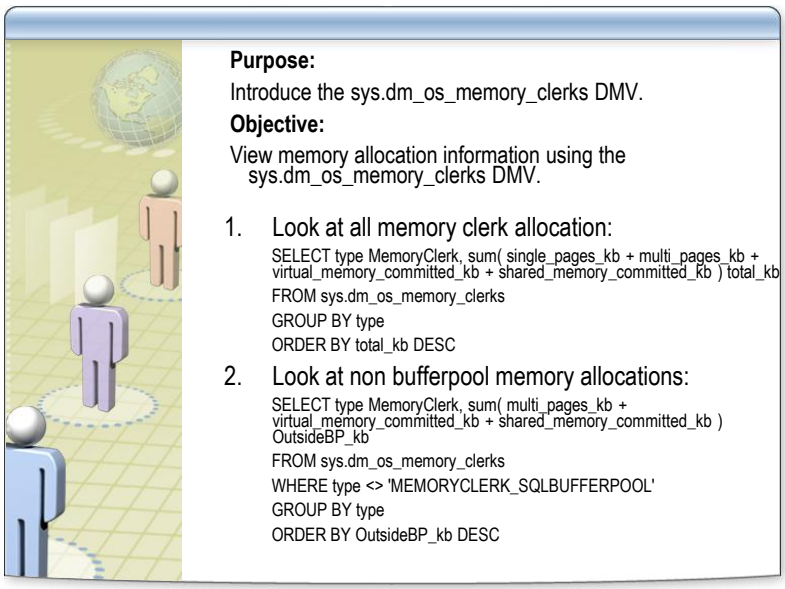

Memory issues can arise from different sources, which make them difficult to handle. Low memory conditions are typically related to one of the following issues:

- **Buffer pool**. The database caching mechanism must grow the buffer pool to improve the response time, but there is no memory remaining to do this.
- **VAS**. Different components fail when trying to use memory outside the buffer pool, because there is no VAS available.
- **Windows memory manager**. The operating system is starting the trimming process to meet the needs of a process that is requesting memory. At this moment, SQL Server needs to discard memory to prevent it from being paged out to disk.

## Resource Monitor

The Resource Monitor performs periodic checks and listens to operating system notifications to determine if the memory is low. When a low memory condition is detected, a message is logged in the Resource Manager Ring buffer. You can use the information in the ring buffer to troubleshoot memory problems. The Resource Manager also broadcasts low memory messages to the Database Engine, so that all internal SQL Server components can reduce their memory consumption, if necessary.

# Demonstration 1: Memory Clerks



**Purpose:**

Introduce the sys.dm_os_memory_clerks DMV.

**Objective:**

View memory allocation information using the sys.dm_os_memory_clerks DMV.

1. Look at all memory clerk allocation:

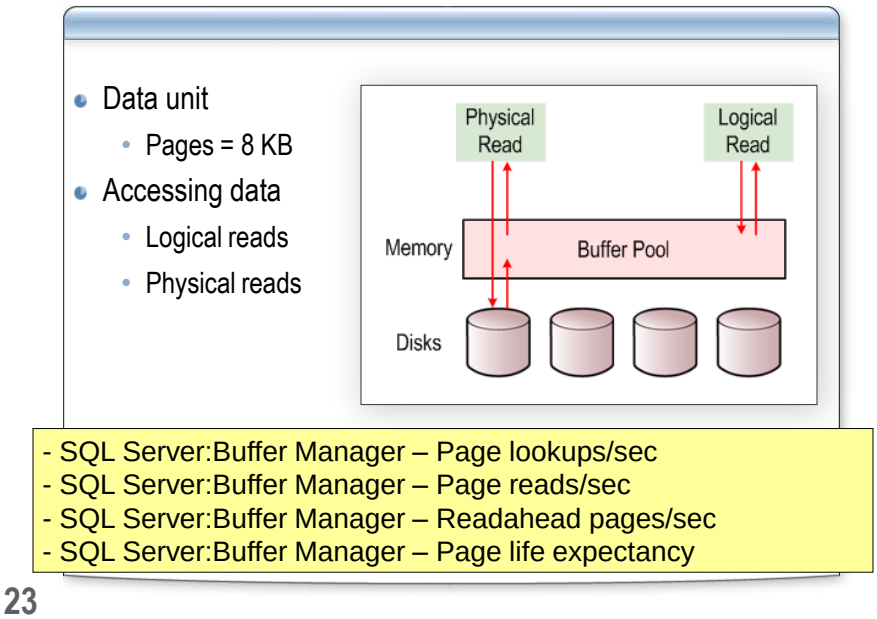
```
SELECT type MemoryClerk, sum( single_pages_kb + multi_pages_kb +
virtual_memory_committed_kb + shared_memory_committed_kb ) total_kb
FROM sys.dm_os_memory_clerks
GROUP BY type
ORDER BY total_kb DESC
```

2. Look at non bufferpool memory allocations:

```
SELECT type MemoryClerk, sum( multi_pages_kb +
virtual_memory_committed_kb + shared_memory_committed_kb )
OutsideBP_kb
FROM sys.dm_os_memory_clerks
WHERE type <> 'MEMORYCLERK_SQLBUFFERPOOL'
GROUP BY type
ORDER BY OutsideBP_kb DESC
```

22

# Monitoring Database Caching




The buffer pool abstracts the physical data location and provides caching functions. All data accessed by SQL Server goes through the buffer pool. Therefore, the buffer pool can determine the data that should be stored in the cache to boost performance for frequently used objects.

There are two ways that the data is accessed from the buffer pool:

- **Logical read**. Indicates that the data requested is already present in the cache. A logical read only involves memory access, so the response time is optimum.
- **Physical read**. Indicates that the data requested is not present in the cache, so the data must be found by using the database files. A physical read is a disk I/O operation, which is more expensive than memory access. When the I/O operation completes, the request is satisfied and the data is stored in the cache so that it can be accessed later.

When a query runs, the database pages that contain the data required to return the query result may or may not be located in physical memory. The query does not have the notion of data locality; it simply requests the page from the buffer pool.

If the pages are in memory, then there is a logical read of the page which results in a ***cache hit***.

If the page is not in memory, then there needs to be a physical read of the page from the disk subsystem. This results in a ***cache miss event***.

## Monitoring the buffer pool

It is important to ensure that the buffer pool is properly caching the data. The easiest way to review the performance of the buffer pool is by using Performance Monitor (Perfmon). The following table describes some of the SQL Server Buffer Manager performance objects that you can use to monitor the buffer pool:

| SQL Server Buffer Manager Counter | Description |
|---|---|
| Page lookups/sec | Counts the number of pages that were requested in that second. This includes both logical and physical reads. In general, this number does not tell much on its own. You should normally compare it with other performance counters. |
| Page reads/sec and Readahead pages/sec | Correspond to the I/O operations posted to the disks. You should add the values of these counters together to determine the total number of pages gathered with physical reads, and then determine if SQL Server is driving high disk utilization. You can also divide this number by Page lookups/sec to determine the cache hit ratio. |
| Page life expectancy | Shows how long a page stays in the buffer if the page is not referenced by a query. A low number usually means that the page stays in the memory for a brief period of time. Therefore, it is important to make sure that this number never stays consistently low. |

**Tip**: The buffer pool works with 8 kilobytes (KB) pages. This means that one read operation corresponds to 8 KB, and 100 reads correspond to 800 KB. You can multiply the counters **Page lookups/sec**, **Page reads/sec**, and **Readahead pages/sec** by 8 KB to determine the total amount of data requested by SQL Server.

When monitoring the performance of the buffer pool, you should consider the following:

- Most often, it is useless to review **Page lookups/sec** alone. This number may vary and will not give you reliable information.
- You should keep both **Page reads/sec** and **Readahead pages/sec** within a low threshold. Doing so will ensure that SQL Server does not stress the disk subsystem.
- The **Readahead pages/sec** counter corresponds to the pages retrieved by using table or index scans. Unless the server is running heavy reports, this is usually a good opportunity to find resource-consuming queries and add appropriate indexes.
- The data pages are written in the background, so the **Pages writes/sec** counter is typically not a concern.
- The **Page life expectancy** counter drops to a lower number every time the pages are being removed. If the value of this counter remains near zero for a long time, it is a clear indication of memory pressure.

- If both **Page reads/sec** and **Readahead pages/sec** remain at zero, it indicates that the buffer pool is maintaining optimum performance. In this scenario, adding memory will not increase the performance related to database caching.

## Monitoring the buffer pool and IO

The following query returns the number of pages in the buffer pool by database and page type:

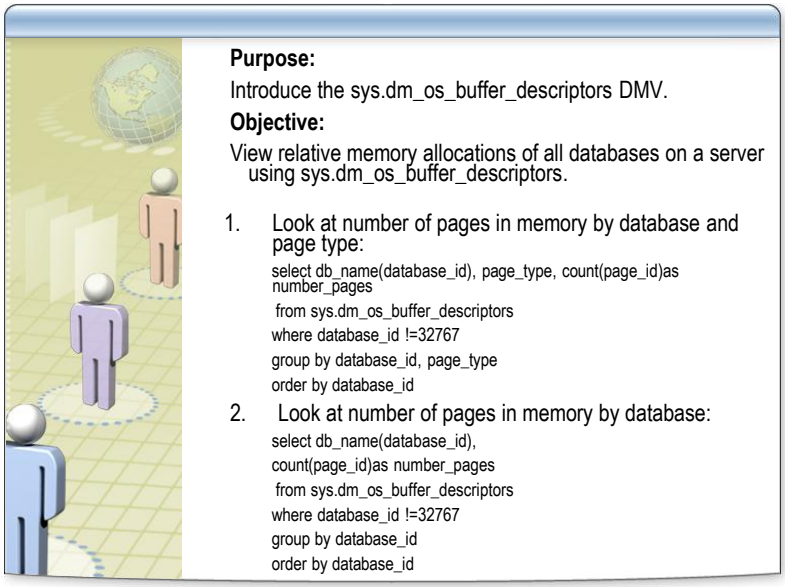
```
select db_name(database_id), page_type, count(page_id)as number_pages
from sys.dm_os_buffer_descriptors
where database_id !=32767
group by database_id, page_type
order by database_id
```

The following query returns the number of pages in the buffer pool by database:

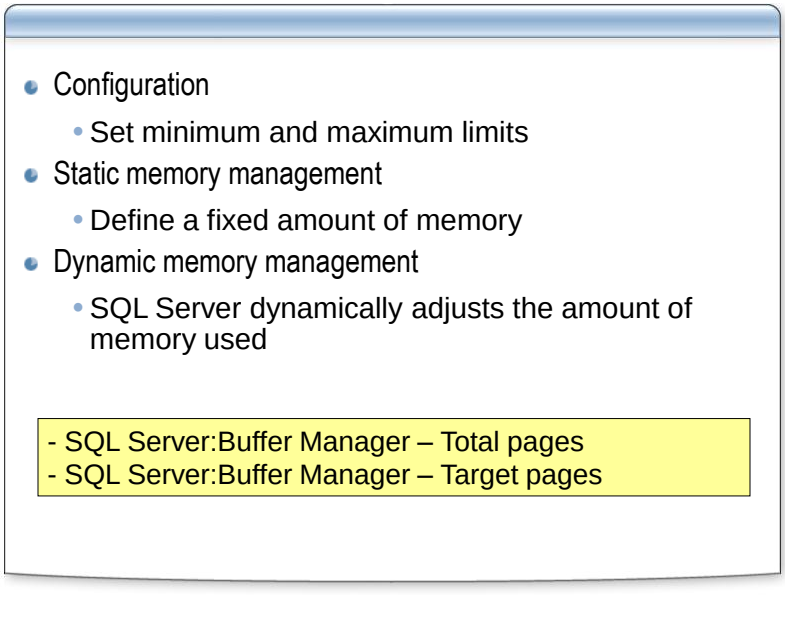
```
select db_name(database_id),
count(page_id)as number_pages
 from sys.dm_os_buffer_descriptors
where database_id !=32767
group by database_id
order by database_id
```

You can use the sys.dm_io_virtual_file_stats DMV to monitor the disk IO statistics for databases and files related to them. You can use the Logical and Physical disk object and counters within System Monitor to build a comprehensive report on disk IO for both capacity planning and performance management.

# Demonstration 2: Memory Distribution



# Monitoring Buffer Pool Size




You can set up the minimum and maximum limits for the total size of the buffer pool by using **sp_configure** with the min server memory and max server memory options. You can change these options at any time without restarting the server.

The SQL Server memory manager can operate in the following two memory management modes:

- **Static memory management**. In this mode, SQL Server uses a fixed amount of memory as defined in **sp_configure**. To enable static memory management, you should set the **min server memory** and **max server memory** options to the same value.
- **Dynamic memory management**. In this mode, the buffer pool size varies in the range that you set by using the **min server memory** and **max server memory** options. SQL Server always tries to achieve the value set in the **max server memory** option unless it receives a notification of memory pressure.

Dynamic memory management enables better resource utilization between SQL Server and the operating system. However, it is always important to continue to monitor the server memory notifications by monitoring the ring buffer. If the ring buffer receives notifications constantly, it may indicate that **max server memory** is set too high.

**Note: The** buffer pool does not immediately acquire the amount of memory specified in **min server memory**. The buffer pool starts with only the memory required for initialization. As the Database Engine workload increases, the buffer pool continues to acquire the memory needed to support the workload. The buffer pool does not free any of the acquired memory until it reaches the amount specified in **min server memory**. When **min server memory** is reached, the buffer pool uses the standard algorithm to acquire and free memory as needed. Once reached, the buffer pool never drops its memory allocation below the level specified in **min server memory**.

## How to monitor the buffer pool size

To maximize database caching, you should have the buffer pool use most of the physical memory that is available.

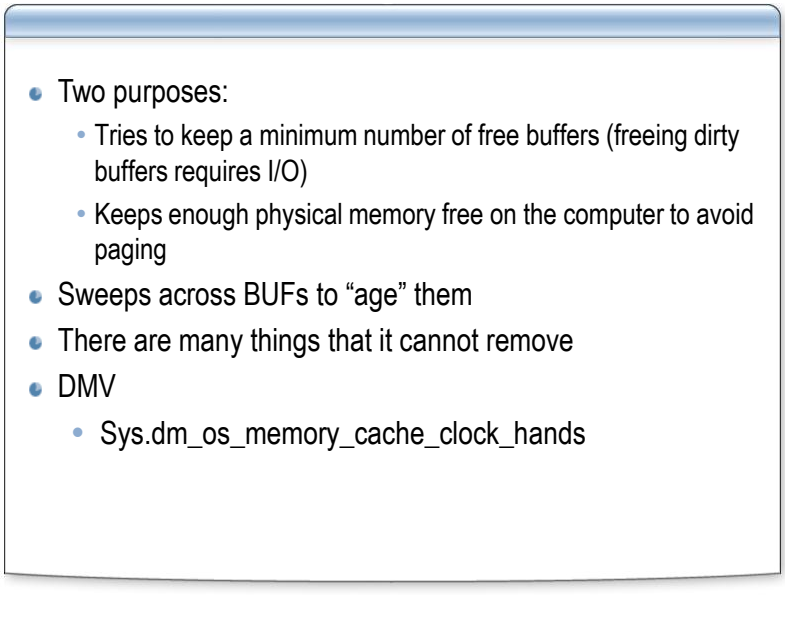
The following table describes some of the SQL Server Buffer Manager performance objects that you can use to monitor the buffer pool size:

| SQL Server Buffer Manager Counter | Description |
|---|---|
| Total pages | Corresponds to the total number of 8 KB pages consumed by the buffer pool. The correct way to check the amount of memory is by looking at this counter rather than the Task Manager, which might report incorrect values for SQL Server because it reports the process working set. |
| Target pages | Specifies the ideal size for the buffer pool as calculated by SQL Server. When calculating this number, SQL Server evaluates the amount of physical memory and VAS.<br>This counter also reacts to memory pressure. For example, when SQL Server is under memory pressure, the value for Target pages decreases accordingly. Afterwards, SQL Server decreases memory consumption until the value for Total pages reaches the value for Target pages. |

When monitoring the size of the buffer pool, you should consider the following:

- Both **Target pages** and **Total pages** should obey the limits set by the **min server memory** and **max server memory** options.
- On a dedicated SQL Server computer, Target pages should be close to the amount of physical memory or to the value of max server memory.
- The Target pages counter is typically limited by VAS in 32-bit computers with more than 2 GB of memory. You will typically see Target pages at either 208,000 pages or 336,000 pages, if the **/3GB** switch is enabled.
- If Target pages consistently decreases over time, it indicates external memory pressure. This can adversely affect SQL Server performance.
- If **Total pages** is lower than Target pages, it indicates that the SQL Server has not yet allocated all the available memory. At this moment, adding more memory to SQL Server will not increase the buffer pool performance.
- The larger the buffer pool, the more effective is the database caching.

# Aging Buffers




SQL Server manages the buffer pool by using the least recently used algorithm. Worker threads can age buffers out of the cache. SQL Server also has dedicated threads called the lazywriter, which help manage the buffer pool.

The lazywriter process sleeps for an interval of time. When it is restarted, it checks the size of the free buffer list. If the free buffer list is below a certain point (based on the size of the cache), the lazywriter process scans the buffer cache to reclaim unused pages and write dirty pages that have a reference count of zero. This makes the reclaimed buffer pages available to new data pages, or for use by the SQL Server or Windows memory managers.

The lazywriter process also removes an object if the memory manager needs memory, all available memory is currently in use, and the age field for the object is zero.

The lazywriter cannot remover buffers with any of the following attributes:

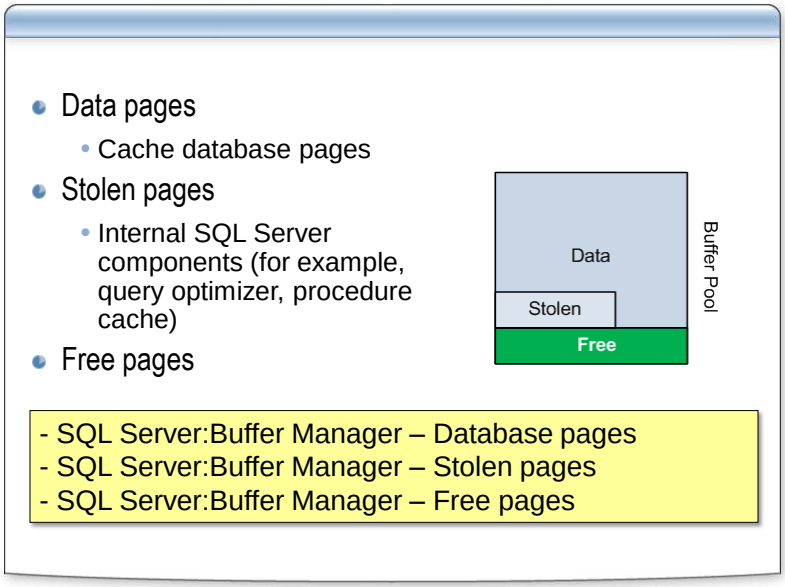
- Any buffer that is currently latched. This means that a user is currently reading or modifying the page.
- A buffer indicating that an I/O is in progress.
- Dirty buffers whose corresponding log records have not been written to the disk. Remember that SQL Server maintains a write-ahead log, and the log pages must be flushed to disk before the associated data pages to ensure the ability to recover the database to a consistent state.

- When the lazywriter finds an eligible buffer, it determines if the buffer is dirty. If it is not dirty, it can simply be unhashed and freed. Otherwise, an asynchronous write is posted so that the buffer can be freed up later.
- On systems with AWE enabled, the buffer might be unmapped (even if it is dirty) from the address space rather than be freed.

The lazywriter always checks 16 buffers at a time. When sufficient free buffers are available, the lazywriter goes to sleep until the next check a second later. If it must resume its search for free buffers, it starts off at the point that it last stopped. When it reaches the end of the buffer pool, it ***wraps*** back around to the beginning and starts over. This is why it is sometimes referred to as a clock, *sweeping* through the buffers in a cyclical fashion.

If the system is running with AWE enabled, the lazywriter works aggressively to limit the count of stolen buffers to 75 percent of the address space.

# Monitoring Memory Distribution




SQL Servers Buffer Pool pages belong to three broad groups. This topic introduces these three groups. Data pages contain data. Stolen pages are buffer pool pages "stolen" from the available data page pool. Pages that are not used for data or "stolen" are Free.

## Data pages

The buffer pool consumes as much memory as it is allowed, and this memory is primarily used for caching data pages. Sometimes, the buffer pool performance is not optimal, even though it has already achieved its maximum size (**Total pages** = **Target pages**). At this point, there is no room left to grow and the internal buffer pool mechanism starts discarding data pages with little usage.

## Stolen pages

The SQL Server memory manager also relies on the buffer pool to allocate memory to internal components, such as:

- Connection
- Procedure cache
- Query optimizer

In order to satisfy these requests, the buffer pool internally uses its own 8 KB pages. The pages just allocated became unusable for database caching and are considered ***stolen pages***, or pages that are already assigned to a component until they are released.

## Free pages

SQL Server must maintain a minimum amount of free pages. These pages are used for further requests; meanwhile, a background task frees up more pages.

## Monitoring memory distribution

The following table describes some of the SQL Server Buffer Manager performance objects that you can use to monitor memory distribution:

| SQL Server Buffer Manager Counter | Description |
|---|---|
| Database pages | Shows the number of pages available for database caching |
| Stolen pages | Shows the number of stolen pages. |
| Free pages | Shows the number of free pages |

When monitoring the size of the buffer pool, you should consider the following:

- You should compare the **Database pages** and **Total pages** counters to make sure that the buffer pool is used mostly for database caching.
- SQL Server might display the number of **Free pages**, varying between 100 and 5,000. This is normal.
- A high number of **Free pages** indicate that the server has plenty of memory. Adding physical memory will not improve the database caching performance.
- A high number of **Stolen pages** do not necessarily indicate a problem. If a server needs a huge procedure cache or a large amount of memory for query execution, it will consume **Stolen pages**. The next step is to investigate the memory distribution among the memory clerks.

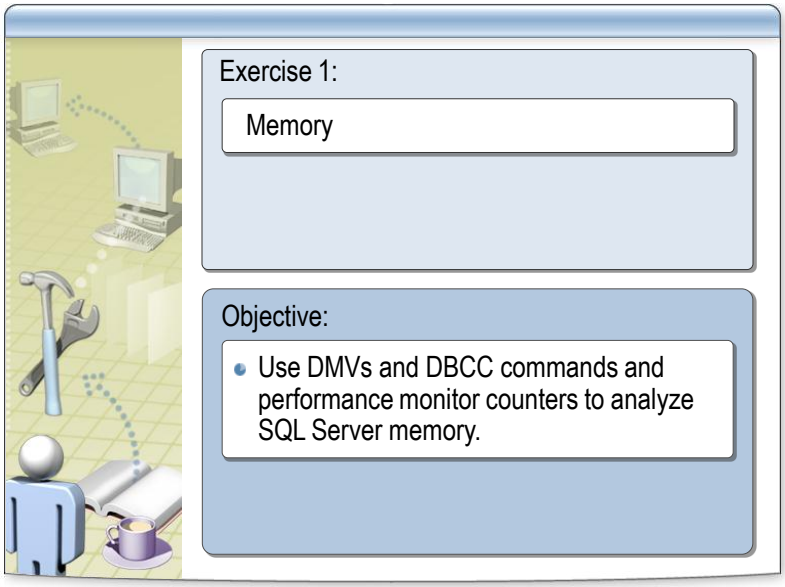
> **Note**: Memory clerks are discussed in more detail later in this module.

# Section 3 Review

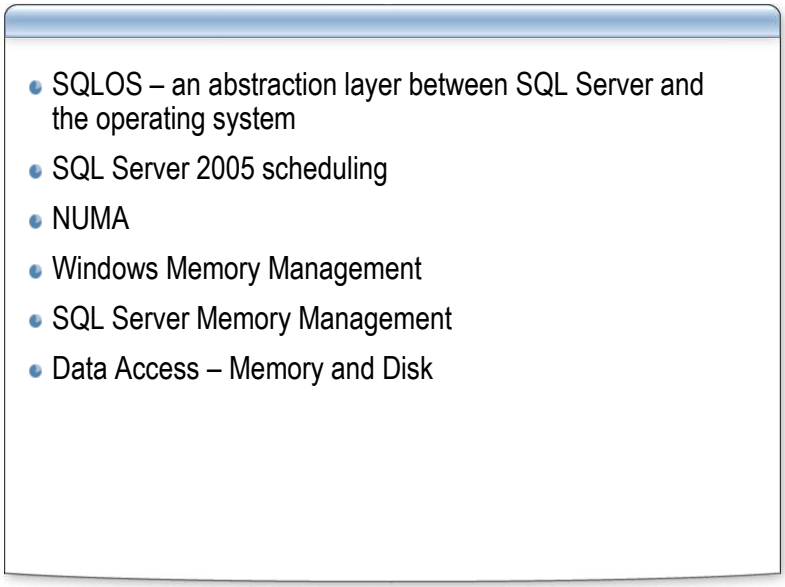
- What is the role of memory clerks in SQL Server?
- Differentiate between logical reads, physical reads, and read aheads.
- Which objects use *stolen pages* within the SQL Server buffer space?

28

# Lab 1: Exercise 1



# Module Summary




This module presented information relating to the architecture of both Windows memory and SQL Server, including how they relate to Non-Uniform Memory Access hardware architectures.

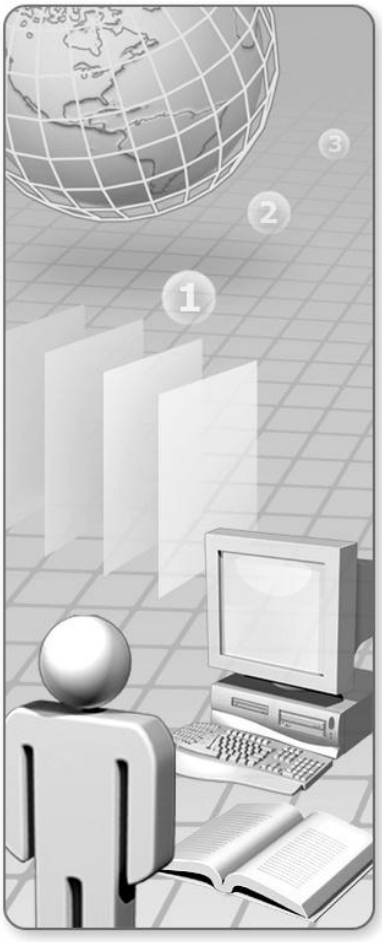
SQL Server 2005 introduces the SQL Server Operating System (SQLOS) to make memory management for SQL Server more efficient. It also makes it easier for SQL Server 2005 and future versions to be adapted to changing hardware architectures.

The SQL Server schedulers have been improved to allow for dynamic worker assignment between the schedulers and allow for the dynamic affinitization of processors.

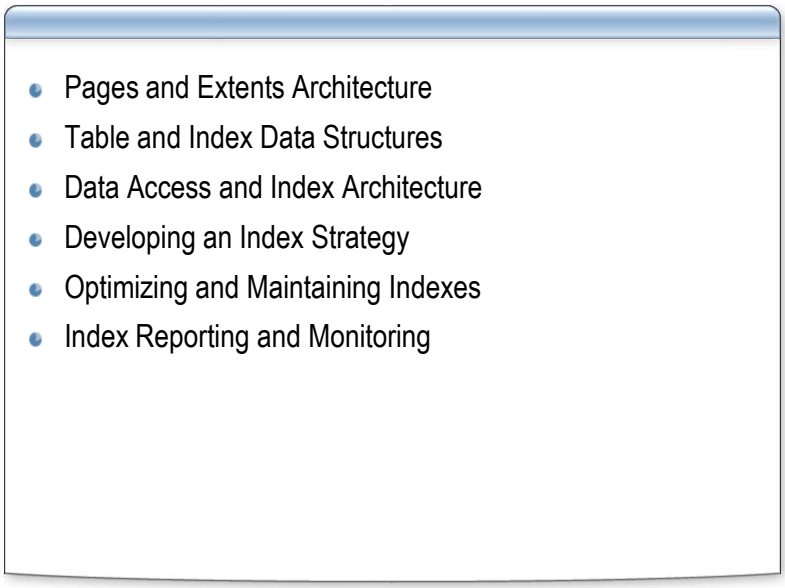
A thorough understanding of these improvements and architectural elements sets a solid foundation for the rest of this workshop.

Microsoft | Services

# Module 2: Table and Index Structure



# Module Overview




## Introduction

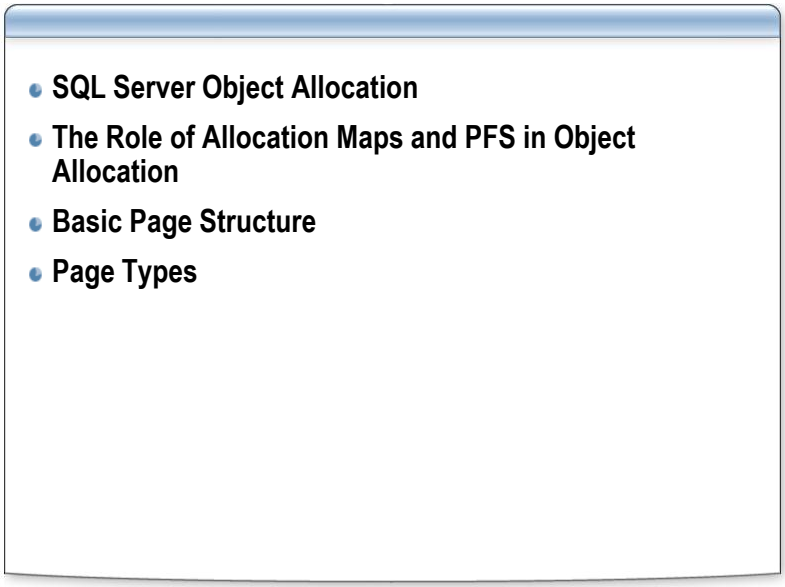
This module describes table and index structures. The module explains how SQL Server allocates and tracks objects on a physical level, and how it logically organizes that data for efficient access by using indexes. You will learn about the various index types and options as well as table and index options. You will then learn how to maintain indexes for optimal performance and take a look at how SQL uses these indexes. The module will introduce the new indexing features made available in SQL Server 2005 and 2008, and explain how you can work with SQL to determine an optimal indexing strategy for a load and also locate missing indexes.

## Objectives

After completing this module, you will be able to:

- Explain the basic architecture of extents and pages in SQL Server.
- Explain the index data structures for a table.
- Explain how different index types and combinations of index types impact data access.
- Develop an indexing strategy that is optimal for a load.
- Optimize and maintain indexes.
- Monitor the indexes in a table.

# Section 1: Pages and Extents Architecture




## Introduction

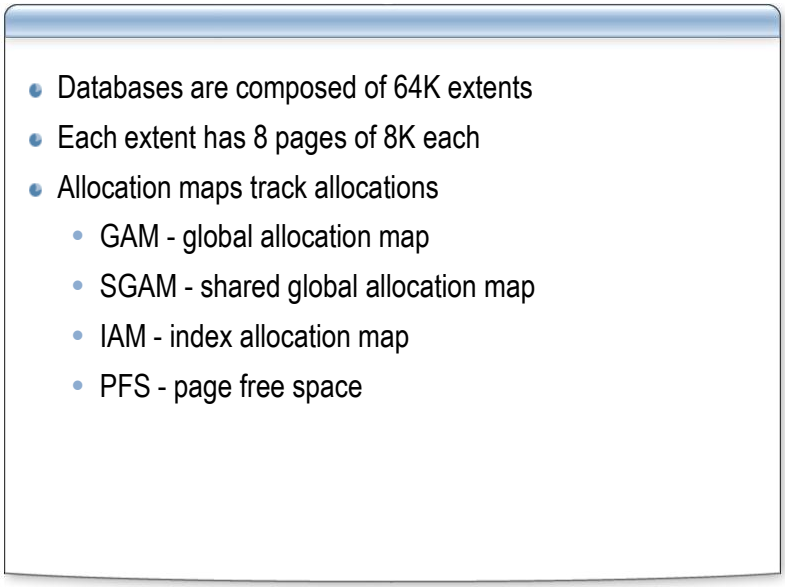
In order to efficiently access just the data that you want, SQL must use a random access file. This allows SQL to refer to a specific data item in a specific location on a disk and read that data without needing to read from the beginning to that point. In order for a random access file to work, it must have a structure of consistently sized allocation units. SQL accomplishes this by using **pages** and **extents**. A system of accounting for pages and extents is established by using *Allocation Maps* and *Page Free Space* pages.

## Objectives

After completing this section, you will be able to:

- Explain how SQL Server uses Global Allocation Maps (GAMs), Shared Global Allocation Maps (SGAMs), and Index Allocation Maps (IAMs) to record the allocation of extents.
- Explain the role of IAM and Page Free Space (PFS) in object allocation.
- Identify the key elements of a basic page structure.

# SQL Server Object Allocation




## Databases are composed of 64 KB extents

**Extents are the basic unit used by SQL Server to manage space**. An extent is eight physically contiguous pages, or 64 kilobytes (KB). SQL Server has two types of extents:

- **Uniform extents** are owned by a single object, and only the owning object can use the eight pages in the extent.
- **Mixed extents** contain eight pages that can be used by different objects.

## Each extent is composed of 8 pages of 8 KB each

To make its space allocation efficient, SQL Server does not allocate an entire extent to tables that contain small amounts of data. A new table or index is usually allocated pages from mixed extents. When the table or index grows to the point that it has eight pages, then all allocations of new space will be done in uniform extents, but the original pages on mixed extents will not be moved into uniform extents. If you create an index on an existing table that has enough rows to generate eight pages in the index, all allocations of new space are uniform extents.

## Allocation maps track allocations

SQL Server uses two types of allocation maps to record the allocation of extents:

- **Global Allocation Map (GAM)**. GAM pages record all extents. Each GAM covers 64,000 extents, or nearly 4 gigabytes (GB) of data. The GAM has one bit for each extent in the interval that it covers. If the bit is 1, the extent is free; if the bit is 0, the extent is allocated.

- **Shared Global Allocation Map (SGAM)**. SGAM pages record those extents that are currently used as mixed extents and have at least one unused page. Each SGAM covers 64,000 extents, or nearly 4 GB of data. The SGAM has one bit for each extent in the interval that it covers. If the bit is 1, the extent is being used as a mixed extent and has free pages; if the bit is 0, the extent is not used as a mixed extent, or it is a mixed extent whose all pages are in use.

## Index Allocation Map (IAM) pages

The IAM pages list the single page allocations for pages stored in mixed extents, and extent allocations for uniform extents used by a table or index. The storage engine can read the IAM to build a sorted list of disk addresses that must be read. The first row of the first IAM in the IAM chain contains a list of singly allocated pages.

There can be a maximum of eight singly allocated pages for any one object. This is because after eight pages are allocated, only uniform extents will be allocated to that object in the future. An IAM page has a header record that indicates the starting extent of the range of extents mapped by the IAM within a partition. The IAM also has a large bitmap, where each bit represents one extent. The first bit in the map represents the first extent in the range, the second bit represents the second extent, and so on. If a bit is 0, the extent that it represents is not allocated to the object owning the IAM. If the bit is 1, the extent that it represents is allocated to the object owning the IAM page.

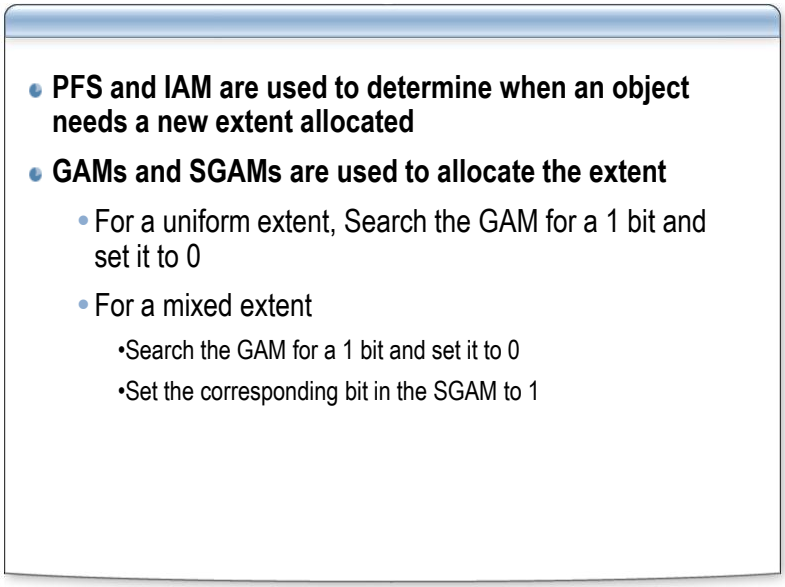
## Page Free Space (PFS) pages

The PFS pages record the allocation status of each page, whether an individual page has been allocated, and the amount of free space on each page. The PFS has one byte for each page recording whether the page is allocated, and if so, whether the page is empty, 1 to 50 percent full, 51 to 80 percent full, 81 to 95 percent full, or 96 to 100 percent full.

After an extent has been allocated to an object, the database engine uses the PFS pages to record which pages in the extent are allocated or free. This information is used when the database engine needs to allocate a new page. The amount of free space in the page is maintained only for heap and text or image pages. The PFS is used when the database engine must find a page in a heap with free space available to hold a newly inserted row. Indexes do not require the free space in the pages to be tracked, because the point at which a new row should be inserted is determined by the index keys.

A PFS page is the first page after the file header page in a data file and is marked as page1. Following the PFS page is a GAM, marked as page 2, followed by an SGAM, marked as page 3. There is a new PFS page every 8,000 pages, approximately. There is a new GAM for each 64,000 extents after the first GAM on page 2, and a new SGAM each 64,000 extents after the first SGAM on page 3.

# The Role of Allocation Maps and PFS in Object Allocation




## PFS and IAM are used to determine when an object needs a new extent allocated

When SQL Server needs to insert a new row, but there is no space available in the current page, SQL Server uses the IAM and PFS pages to find a page with enough space to hold the row. First, SQL Server uses the IAM pages to find the extents allocated to the object. Then, for each extent, SQL Server searches the PFS pages to see if there is a page with enough free space to hold the row. Each IAM and PFS page covers a large number of data pages, which are generally stored in the memory in the SQL Server buffer pool. This means that these pages can be searched quickly.

SQL allocates a new extent to an object only when it cannot quickly find a page in an existing extent with enough space to hold the row that is being inserted. It allocates extents from those available in the filegroup by using a proportional allocation algorithm (files are used in proportion to their free space). Every file in a filegroup should have a similar percentage of space used.

Heaps have one row in **sys.partitions** with **index_id** = 0. The **first_iam_page** column in **sys.system_internals_allocation_units** points to the IAM chain for the collection of heap data pages in the specified partition. The server uses the IAM pages to find the pages in the data page collection.

SQL Server uses the IAM pages to navigate through the heap. Data pages and the rows within them are not in any specific order and are not linked together. The only logical connection between data pages is the connection recorded in the IAM pages.

The IAM pages can be scanned to perform table scans or serial reads of a heap to find the extents holding pages for the heap. Because the IAM represents extents in the same order that they exist in the data files, serial heap scans progress uniformly down each file. Using the IAM pages to set the scan sequence also means that rows from the heap are not typically returned in the order in which they were inserted.

### GAMs and SGAMs are used to allocate the new extent

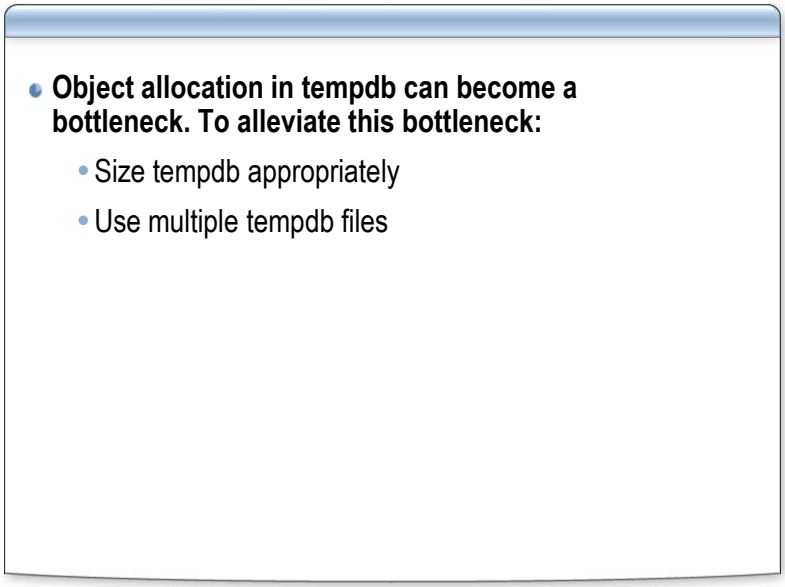
To allocate extents, SQL Server uses one of the following methods:

- To allocate a uniform extent, the database engine searches the GAM for a 1 bit and sets it to 0.
- To allocate a mixed extent, the database engine searches the GAM for a 1 bit, sets it to 0, and then also sets the corresponding bit in the SGAM to 1.

To de-allocate an extent, the database engine ensures that the GAM bit is set to 1 and the SGAM bit is set to 0.

The algorithms that are actually used, internally, by the database engine are more sophisticated than the process described above, because the database engine distributes data evenly in a database.

# Object Allocation Bottlenecks




Only one tempdb exists on each instance of SQL Server, but tempdb is used by all transactions in all databases. Consumers of tempdb include:

- LOB Variables such as VARCHAR(MAX) or NVARCHAR(MAX)
- Temporary tables and table variables
- Common Table Expressions (CTE)
- XML values that will not easily fit into main memory
- Indexes with SORT_IN_TEMPDB
- Inserted and deleted virtual tables in triggers
- Multiple Active Result Sets (MARS)
- Online indexing operations
- Work files for hash joins and hash aggregations
- Work tables for table spool operations
- Row versioning for Snapshot Isolation Level and Read Committed Snapshot

On a busy system, the allocation and de-allocation of such objects can create a bottleneck on tempdb. Tempdb can be optimized with the following steps

### Size tempdb Appropriately

Pre-allocate space for all tempdb files by setting the file size to a value large enough to accommodate the typical workload in the environment. This prevents tempdb from

expanding too frequently, which can affect performance. The tempdb database should be set to "autogrow," but this should be used to increase disk space for unplanned exceptions.

### Use Multiple Data Files

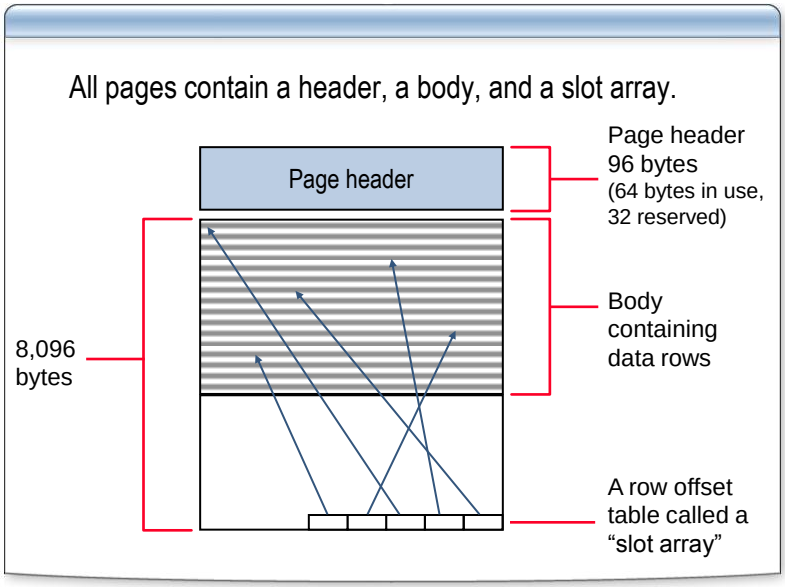
Each file begins with its own file header page, PFS, SGAM and GAM page. One way to reduce contention on these pages for object allocation and deallocation is to create multiple PFS, SGAM and GAM pages. This is done by creating multiple tempdb files. Follow these recommendations when designing your system to use multiple tempdb files:

- As a general guideline, create one data file for each CPU being used by SQL server. Then adjust the number of files up or down as necessary. In most cases, it will not be necessary to use more than 8 tempdb files.

> Note: A dual-core CPU is considered to be two CPUs. For the purposes of estimating the number of tempdb files, a hyperthreaded CPU is considered to be a single processor.

- Make each data file the same size; this allows for optimal proportional-fill performance.
- Put the tempdb database on a fast input-output subsystem. Use disk striping if there are many directly attached disks.
- Put the tempdb database files on disks that differ from those that are used by transaction logs or user databases.

# Basic Page Structure




All SQL Server pages contain a header, a body containing data rows, and a slot array.

## Page header

The page header is the first 96 bytes of each data page. It contains the page header information such as the Page ID, Page type, and Allocation Unit ID of the page.

## Data rows

This section holds the actual data rows of the table. Data rows have the following characteristics:

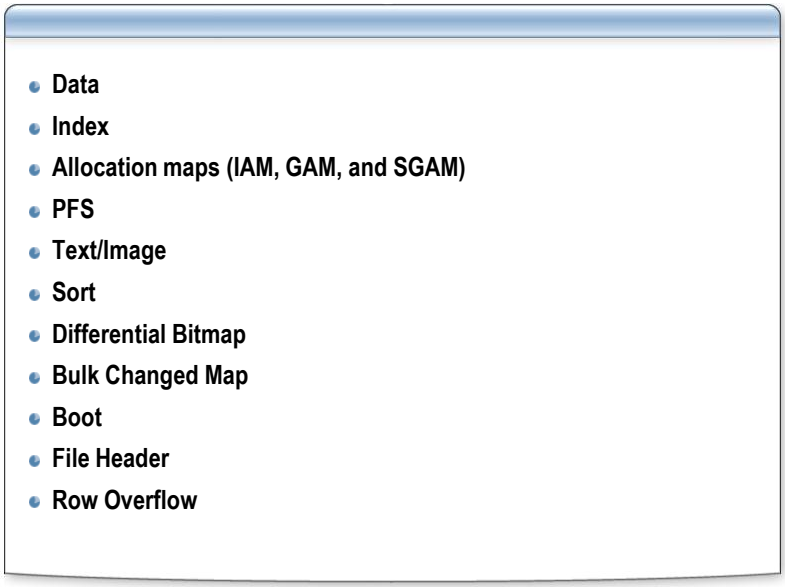
- The maximum size of a single data row is 8,060 bytes.
- A normal data row cannot span multiple pages.
- Text and image data columns can be stored in their own separate pages and can span multiple pages.
- Fixed-length columns always store the same number of rows per page.
- Variable-length rows (rows from tables that are defined by using one or more variable length data types) store as many rows as will fit, based on the actual length of the data entered.
- Shorter row length enables more rows to fit on a page, thereby reducing I/O and improving the cache/hit ratio.

## Row offset array

The row offset array is a block of 2-byte entries, each of which indicates the offset on the page on which the corresponding data row begins. Each row has a 2-byte entry in this array. Although these bytes are not stored in the row with the data, they do have an effect on the number of rows that will fit on a page.

The row offset array indicates the logical order of rows on a page. For example, if a table has a clustered index, the logical order is the order of the clustered index key. This does not mean that the rows will be physically stored on the page in the order of the clustered index key. Instead, slot 0 in the offset array refers to the first row in the order, slot 1 refers to the second row, and so forth. The offset of these rows can be anywhere on the page.

# Page Types

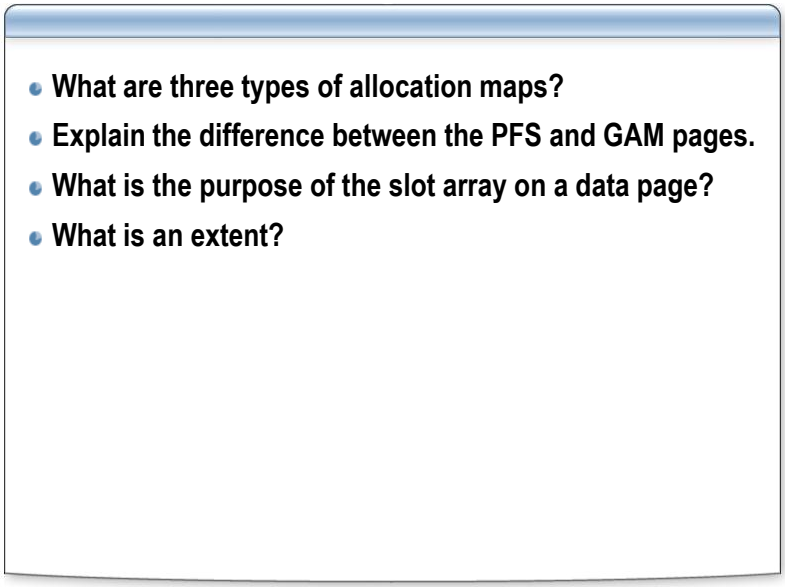



The following table describes the SQL Server page types and the information that they contain:

| Page Type | Contains |
|---|---|
| Data | Data records. |
| Index | Non-clustered index leaf records and non-leaf records from clustered and non-clustered indexes. |
| IAM | Allocation information about extents within a fixed 4-GB GAM interval that are allocated to an index in SQL Server 2000 and to an allocation unit in SQL Server 2005 or 2008. |
| GAM and SGAM | Global allocation information about extents in a GAM interval. |
| PFS | Allocation and free space information about pages within a PFS interval (approximately 64 MB). |
| Text/Image | Two types of pages that hold leaf and intermediate nodes in text trees. |
| Sort | Sort records being used in active sort operations. |
| Differential bitmap | Information about which extents in a GAM interval have changed since the last full or differential backup. |
| Bulk-changed map | Information about which extents in a GAM interval have changed while in bulk-logged mode since the last backup. This information enables you to switch to bulk-logged mode for bulk loads and index rebuilds without worrying about breaking a backup chain. |
| Boot | Information about the database; Each database has only one Boot page. |
| File header | Information about the file. It is the first page (page 0) in every file. |
| Row Overflow | The largest variable length column data from a record when the total length of the record exceeds 8060 bytes. |

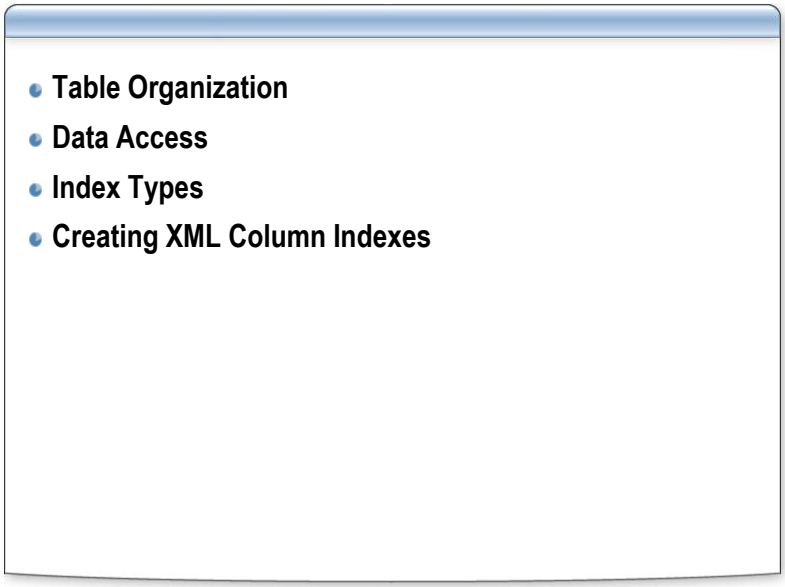
**Note:** Binary Large object pages (Text, Image), frequently called BLOBs, often use separate pages with different structures. For more information, refer to **Using Large-Value Data Types** in *SQL Server 2005 Books Online*.

# Section 1 Review




- What is an extent?
- What are the three types of allocation maps?
- Explain the difference between the PFS and GAM pages.
- What is the purpose of slot array on a data page?

# Section 2: Table and Index Data Structures




## Introduction

Extents and pages are the physical structures that contain the data and metadata in a database. SQL uses allocation maps and PFS pages to allocate and access the data on a physical level, but a logical structure must be built on top of this physical structure for efficiently accessing the data.

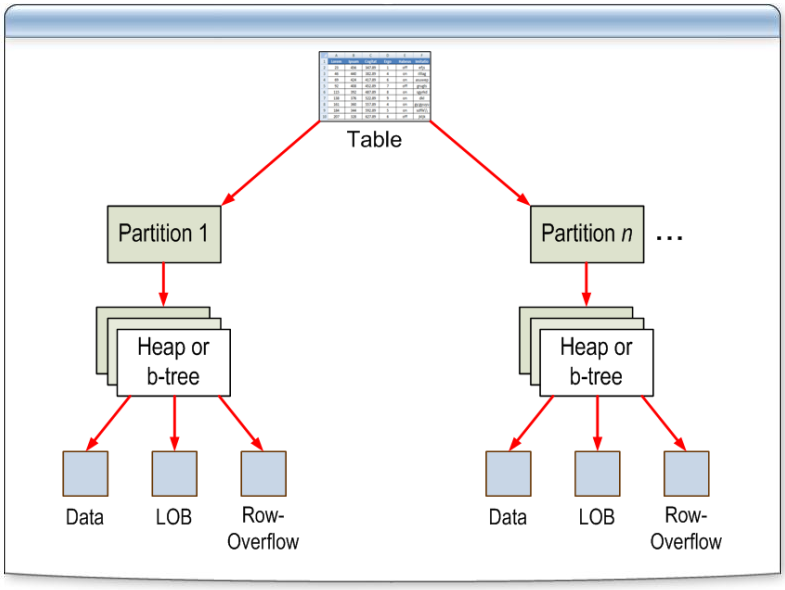
This section describes the logical structures of tables and indexes, and how these structures are used to access data.

## Objectives

After completing this section, you will be able to:

- Explain partitioning and allocation units in table organization.
- Explain how SQL Server uses the information in system views to access data structures, and how you can use them to determine the need for maintenance.
- Differentiate between the various types of SQL Server indexes.
- Create primary and secondary XML column indexes.

# Table Organization




11

A table is contained in one or more partitions and each partition contains data rows in either a heap or a clustered index structure. The pages of the heap or clustered index are managed in one or more allocation units, depending on the column types in the data rows.

## Partitions

A partition is a user-defined unit of data organization. By default, a table or index has only one partition that contains all the table or index pages. A table or index with a single partition is equivalent to the organizational structure of tables and indexes prior to SQL Server 2005.

When a table or index uses multiple partitions, the data is partitioned horizontally, so that groups of rows are mapped to individual partitions, based on specified columns. The partitions can be put into one or more filegroups in the database. The table or index is treated as a single logical entity when queries or updates are performed on the data.

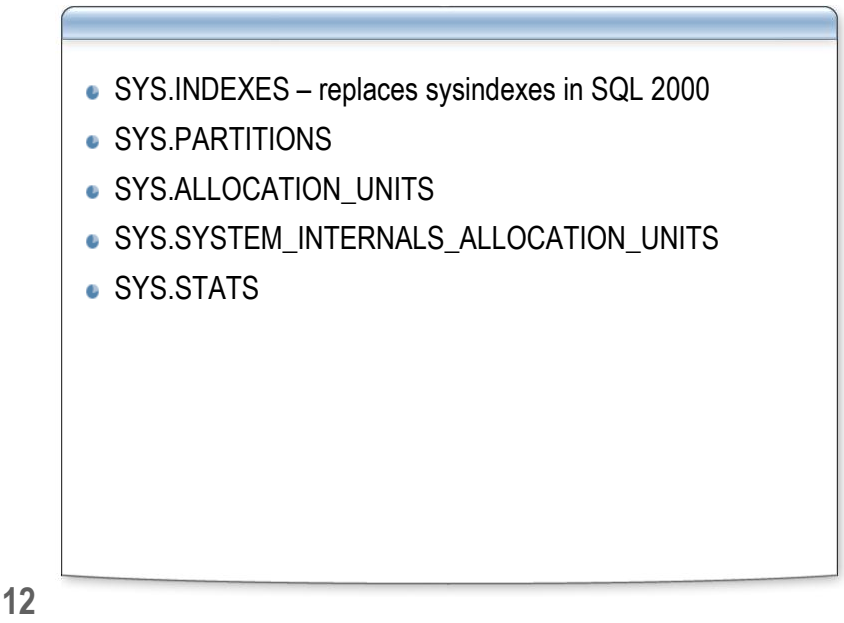
## Allocation units

An allocation unit is a collection of pages within a heap or B-tree (HoBT) used to manage data based on their page type. The following table lists the types of allocation units used to manage data in tables and indexes:

| Allocation Unit | Description |
|---|---|
| IN_ROW_DATA | Data or index rows that contain all data except Large Object (LOB) data. Pages are of type Data or Index. |
| LOB_DATA | Large object data stored in one or more of these data types: **text**, **ntext**, **image**, **xml**, **varchar(max)**, **nvarchar(max)**, **varbinary(max)**, |

| Allocation Unit | Description |
|---|---|
| | or **CLR user-defined types (CLR UDT)**. Pages are of type Text or Image. |
| ROW_OVERFLOW_DATA | Variable length data stored in **varchar**, **nvarchar**, **varbinary**, or **sql_variant** columns that exceed the 8,060-byte row size limit. Pages are of type Data. |

# Data Access




All data is stored in the data files in no particular physical order. In order for SQL to access the data for one particular object, it must contain a starting place, and a way of differentiating the pages and extents belonging to that object from pages and extents belonging to other objects. The following table describes the system views that you can use to see the entry point into each object, as well as information about the structure and contents of that object:

| View Name | Description |
|---|---|
| sys.indexes | Contains one row per heap or index. This is a system view (not a table) and does not have pointers to data. |
| sys.partitions | Contains a row for each partition of all the tables and indexes in the database. All tables and indexes in SQL Server 2005 or later are considered to contain at least one partition, even if they are not explicitly partitioned. The HoBT_id and partition_id columns will be the same, and they will each return one row for each partition that each object is using for storage. |
| sys.allocation_units | Contains a row for each allocation unit in the database. A partition can have up to four different allocation units, one each for:<br>• **DROPPED**<br>• **IN_ROW_DATA**<br>• **LOB_DATA**<br>• **ROW_OVERFLOW_DATA**<br>The **first_iam_page** column points to the chain of IAM pages that manage the pages in the LOB_DATA allocation unit. |

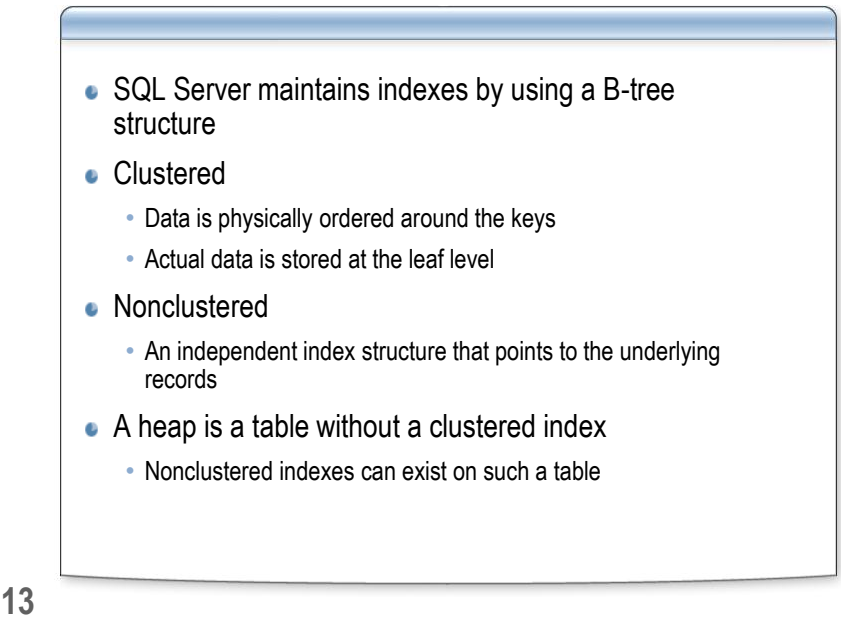
| View Name | Description |
|---|---|
| sys.system_internals_ allocation_units | Contains columns with the pointers for **first_iam_page** and root pages of an allocation unit. In previous releases, the **firstiam** column and root pages were stored in sysindexes. |
| sys.stats | Contains a row for each statistic. Each index has a corresponding statistics row with the same name and ID (**index_id = stats_id**), but not every statistics row has a corresponding index. In the previous versions, statistics were inserted in sysindexes.<br>Entries with names beginning with **_WA_sys_** are only statblobs representing the statistic spread of the data pages. Statblobs are maintained by SQL Server on all indexes and all CREATE STATISTICS commands that you create on a specific column, or that are created automatically (auto statistics) by the database, when you restrict a column that does not contain an index. The statsblob is a binary data sampling of an index, and helps the optimizer choose the best type of navigation to service a query. |

SQL Server tracks which pages belong to a table or index by using IAM pages. The IAM enables SQL Server to do efficient prefetching of the table extents, but every row must be scanned.

## Try This

```
USE AdventureWorksPTO
go

select * from sys.indexes where object_id=object_id('person.address')
select * From sys.stats where object_id = object_id('person.address')

-- to get a list of partitions and data pages used for a table run this query
SELECT o.name AS table_name, p.index_id, i.name AS index_name , au.type_desc AS
allocation_type, au.data_pages, partition_number
FROM sys.allocation_units AS au
    JOIN sys.partitions AS p ON au.container_id = p.partition_id
    JOIN sys.objects AS o ON p.object_id = o.object_id
    JOIN sys.indexes AS i ON p.index_id = i.index_id AND i.object_id = p.object_id
ORDER BY o.name, p.index_id;
```

# Index Types




SQL Server indexes are organized as B-trees. Each page in an index holds a page header followed by index rows. Each index row contains a key value and a pointer to either a lower-level page or a data row. Each page in an index is called an *index node*. The top node of the B-tree is called the *root node*. The nodes on the bottom layer of the index are called the *leaf nodes*. The pages in each level of the index are linked together in a doubly-linked list. In a clustered index, the data pages make up the leaf nodes. Any index levels between the root and the leaves are collectively known as *intermediate levels*.

SQL Server has two types of indexes—clustered and non-clustered.

## Clustered indexes

In a clustered index, the data rows are stored in order, based on the clustered index key. The leaf level of the clustered index is the actual table that contains the data pages. The index is implemented as a B-tree index structure that supports fast retrieval of the rows, based on their clustered index key values. The pages in each level of the index, including the data pages in the leaf level, are linked in a doubly-linked list, but navigation from one level to another is done by using the pointers to the child pages associated with the key values.

A clustered index on a table or a view has a row in **sys.partitions** with **index_id** = 1.

The **root_page** column in **sys.system_internals_allocation_units** points to the top of the clustered index B-tree in the specified partition. The server uses the index B-tree to find the data pages in the partition.

Because the actual page chain for the data pages can be ordered in only one way, a table can have only one clustered index. The query optimizer strongly favors a clustered index for seeks, because such an index enables the data to be found directly at the leaf level. Because it defines the actual order of the data, a clustered index allows especially fast access for queries looking for a range of values. The query optimizer detects that only a certain range of data pages must be scanned.

### Most tables should have a clustered index

If a table has only one index, it generally should be clustered. Many documents describing SQL Server indexes will tell you that the clustered index physically stores the data in sorted order. This can be misleading, if you think of physical storage as the disk itself. If a clustered index had to keep the data on the actual disk in a particular order, it would be prohibitively expensive to make changes. If a page got too full and had to be split into two, the data on all the succeeding pages would have to be moved down. Sorted order in a clustered index simply means that the data page chain is in order. If SQL Server follows the page chain, it can access each row in clustered index order, but new pages can be added by simply adjusting the links in the page chain.

### All clustered indexes are unique

If you build a clustered index without specifying the unique keyword, SQL Server forces uniqueness by adding a uniqueifier to the rows, when necessary. This uniqueifier is a 4-byte value added as an additional sort key to only those rows that have duplicates of their primary sort keys.

### Keep clustered indexes narrow

Keeping your clustered index key value small increases the number of index rows that can be placed on an index page. This decreases the number of levels that must be traversed, thereby improving performance by reducing I/O. Also, the clustered index key is duplicated in every non-clustered index row, so keeping the clustered key small decreases the space required in all non-clustered indexes.

## Non-clustered indexes

A non-clustered index in SQL server will have a row in sys.partitions with index_id between 2 and 254.

Non-clustered indexes have the same structure as clustered indexes, with two significant differences:

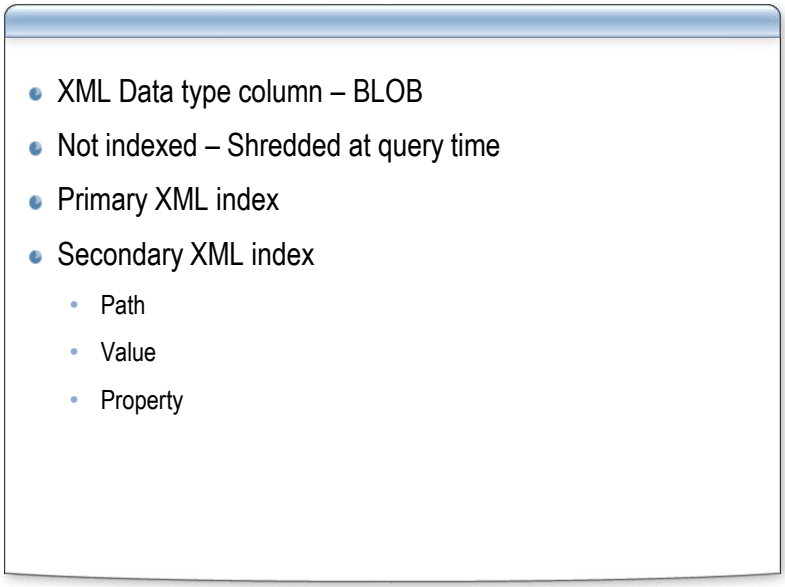
- The data rows are not sorted and stored in order based on non-clustered index keys.
- The leaf level of a non-clustered index does not consist of the data pages. The leaf level stores the index key, included columns, and:
  - A Row Identifier (RID), if no clustered index exists on the table. The RID is an 8-byte binary value containing the fileid, pageid, and slotid on the page of the indexed table.
  - The clustered index keys, if the table contains a clustered index.

The **root_page** column in **sys.system_internals_allocation_units** points to the top of the non-clustered index B-tree in the specified partition.

### Heap

A heap in SQL Server is a table that does not have a clustered index. A heap has a row in **sys.partitions** with **index_id** = 0.

# Creating XML Column Indexes




## Primary XML index

The primary XML index is a shredded and persisted representation of the XML binary large objects (BLOBs) in the **xml** data type column. For each XML BLOB in the column, the index creates several rows of data, approximately one row for each node in the XML. Each row stores the following:

- Name of the attribute (Tag Name) (int)
- Node value (int)
- Node type (Element, Attribute, Text)
- Document order
- Path from node to the XML root
- Clustered Index key of the base table (similar to all non-clustered indexes)

The following example shows how to create a primary XML index:

```
USE AdventureWorksPTO;
GO
IF EXISTS (SELECT * FROM sys.indexes
WHERE name = N'PXML_ProductModel_CatalogDescription')
DROP INDEX PXML_ProductModel_CatalogDescription
ON Production.ProductModel;
GO
CREATE PRIMARY XML INDEX PXML_ProductModel_CatalogDescription
ON Production.ProductModel (CatalogDescription);
GO
```

## Secondary XML index

Secondary XML indexes can further enhance performance, but a primary XML index must first exist before you can create secondary indexes. The following table describes the purpose of the three types of secondary index:

| Secondary XML Index Type | Purpose |
|---|---|
| PATH | Enhances performance where queries specify path expressions on XML type columns. |
| VALUE | Enhances performance of value-based queries or where the path contains a wildcard. Key columns of the VALUE index are node value and path of the primary XML index. |
| PROPERTY | Enhances performance of queries that retrieve one or more values from individual XML instances. This index is built on columns PK, path, and node value of the primary XML index (PK is the primary key of the base table). |

The following example shows how to create a secondary XML index:

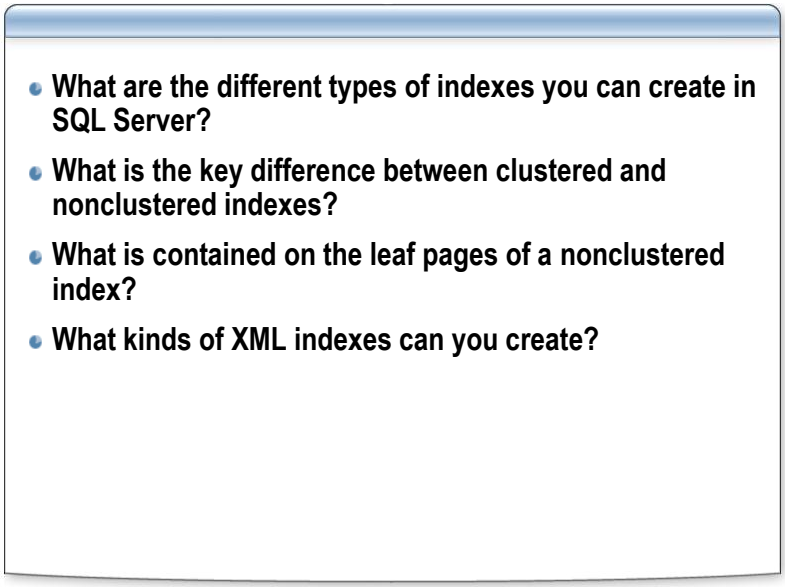
```
USE AdventureWorksPTO;
GO
IF EXISTS (SELECT name FROM sys.indexes
WHERE name = N'IXML_ProductModel_CatalogDescription_Path')
DROP INDEX IXML_ProductModel_CatalogDescription_Path
ON Production.ProductModel;
GO
CREATE XML INDEX IXML_ProductModel_CatalogDescription_Path
ON Production.ProductModel (CatalogDescription)
USING XML INDEX PXML_ProductModel_CatalogDescription FOR PATH ;
GO
```

## Discovery of XML indexes

You can view the XML indexes that exist in your database by querying sys.xml_indexes as follows:

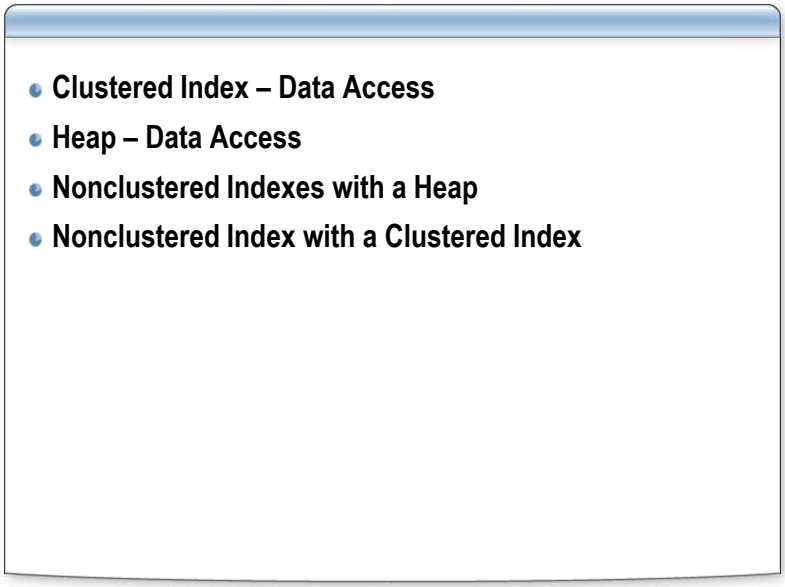
```
Select using_xml_index_id, secondary_type, secondary_type_desc from
sys.xml_indexes
```

# Section 2 Review




- What are the different types of indexes that you can create in SQL Server?
- What is the key difference between clustered and non-clustered indexes?
- What is contained on the leaf pages of a non-clustered index?
- What are the different types of XML indexes that you can create?

# Section 3: Data Access and Index Architecture




## Introduction

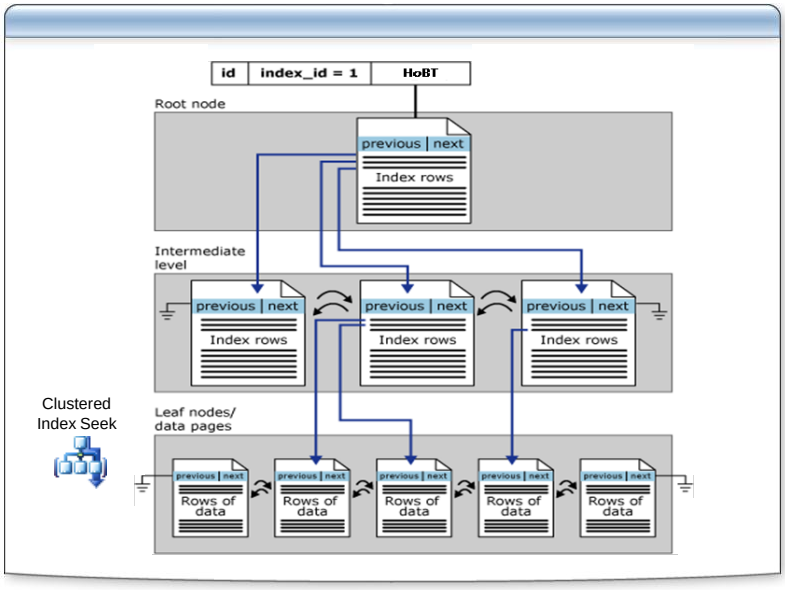
Each data structure and index type comes with its own unique set of algorithms for accessing data. This section explains how SQL server accesses data by using clustered indexes, heaps, non-clustered indexes with heaps, and non-clustered indexes with clustered indexes.

## Objectives

After completing this section, you will be able to:

- Explain how SQL Server accesses tables with a clustered index.
- Explain how SQL Server accesses tables with a heap.
- Explain how SQL Server accesses tables with both a non-clustered indexes and a heap.
- Explain how SQL Server accesses tables with both a non-clustered index and a clustered index.

# Clustered Index—Data Access




The illustration on the slide above shows data access by using a clustered index. A *clustered index seek* is the process of starting at the root node, reading the keys until the correct value is located, then following the pointers to the child_fileid and child_pageid to the child page until the root page that contains the first value requested is reached. If more rows are requested in the range than are contained on one leaf page, then the pointer to the next leaf page is followed until the entire range desired has been retrieved.

When a query requests multiple disjoint ranges (for example, SELECT * from Tab1 WHERE col1 BETWEEN 1 and 3 OR col1 BETWEEN 10 AND 20), the B-tree will be navigated from root to the leaf page that contains the beginning point of the first range. The doubly linked list will then be followed along the leaf level until the end of the range is reached. The B-tree will then be navigated from root to leaf for the beginning point of the next range, and again, the doubly linked list will be followed at the leaf level to the end of the second range. This process is repeated for each disjoint range in the query.

> **Note**: The process explained above will be discussed further in the ***Query Optimization*** module.

For a clustered index scan, the B-tree is navigated from the root to the first logical page of the leaf level. Each page of the leaf level is then read and the next leaf page is found by following the pointers in the doubly linked list at the leaf level until the last leaf page is reached.

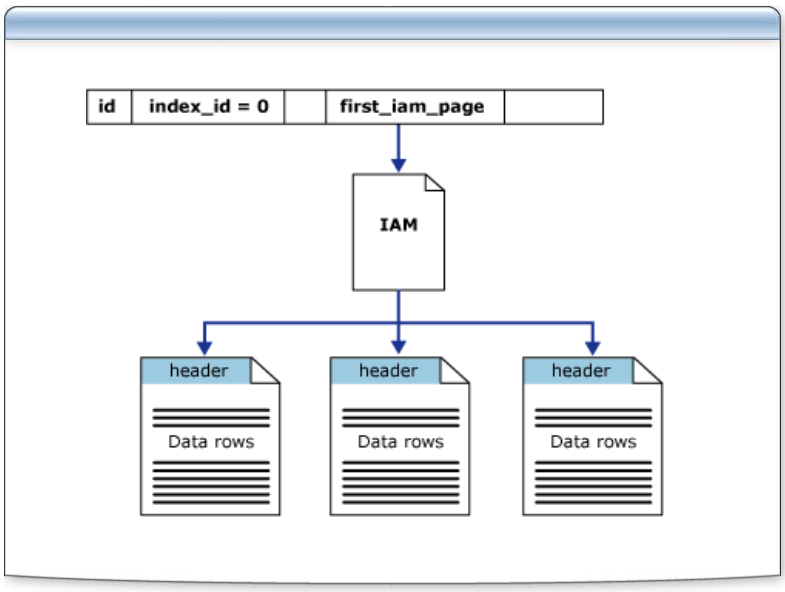
Note that the **index_id** in the **sys.partitions** table is set to 1 for clustered indexes. The example shown on the slide above includes one intermediate level, though there can be many such levels, depending on the number of rows in the table and the width of the index key. The leaf level of the index includes the data pages of the base table.

The following query returns information about the clustered indexes in the current database:

```
SELECT o.name AS table_name, p.index_id, i.name AS index_name, au.type_desc AS
allocation_type, au.data_pages, partition_number, au.root_page
FROM sys.system_internals_allocation_units AS au
    JOIN sys.partitions AS p ON au.container_id = p.partition_id
    JOIN sys.objects AS o ON p.object_id = o.object_id
    JOIN sys.indexes AS i ON p.index_id = i.index_id AND i.object_id = p.object_id
and p.index_id=1
ORDER BY o.name, p.index_id;
```

**Note**: The **au.root** value is the page number of the root node of the index.

## Heap—Data Access




The illustration on the slide above shows data access by using a heap. Note that the **index_id** in the **sys.partitions** table is set to 0 for heaps. There is no intermediate level shown in the example above because there can be no intermediate level for a heap.

A table scan is the only data access method that uses the heap directly. For a table scan, the first IAM page is read. The first IAM page contains a maximum of 8 single page allocations on its first row. After this, the IAM page indicates the ranges of extents that are allocated to the owning table. If the table is large enough to have more than 1 IAM, then each IAM also contains the pointer to the next IAM page for this object.

When a table scan is performed, the single page allocation pointers are followed to each of these pages. Then, the pointers to the beginning of each range of extents allocated to this table are followed. The PFS contains the information on each page within each extent and whether or not that page contains data. The IAM and PFS are used together to locate each page with data, and those pages are then scanned.

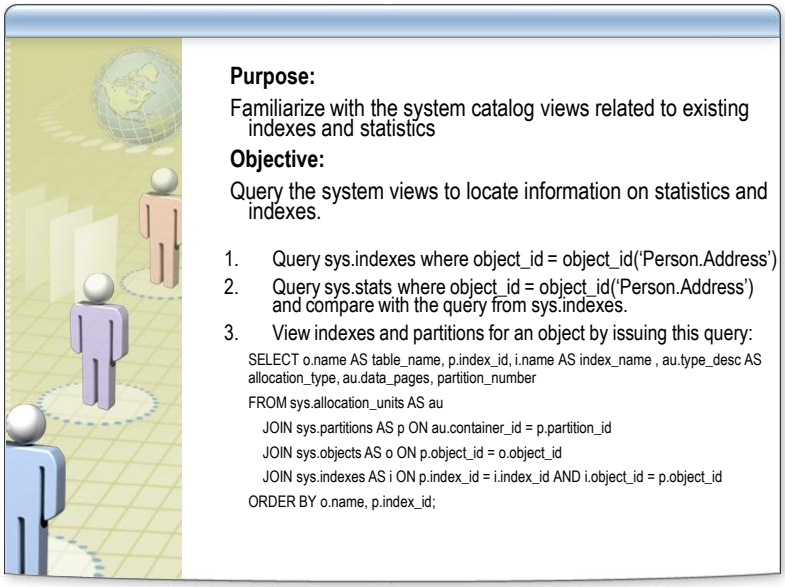
Note that without an index on a table, the entire table must be scanned in this manner even if the query will return only one row.

The following query returns information about the heaps in the current database:

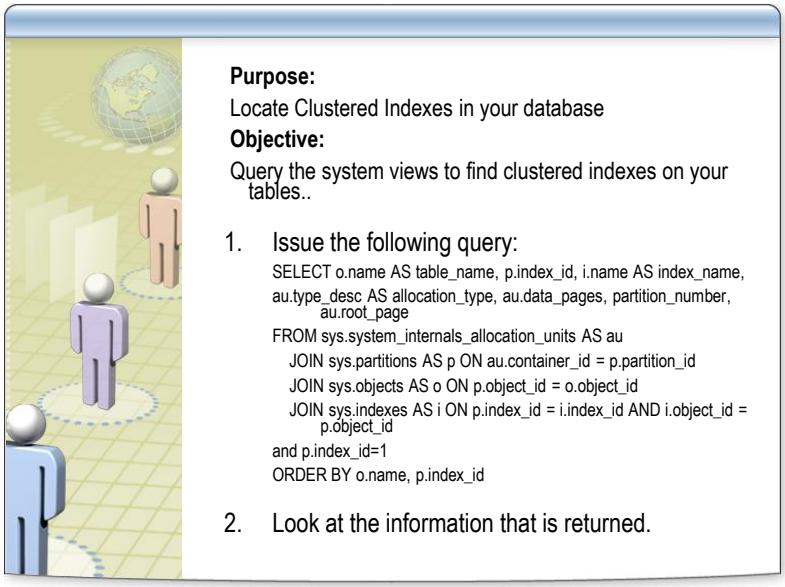
```
SELECT o.name AS table_name, p.index_id, i.name AS index_name, au.type_desc AS
allocation_type, au.data_pages, partition_number, au.first_iam_page
FROM sys.system_internals_allocation_units AS au
    JOIN sys.partitions AS p ON au.container_id = p.partition_id
    JOIN sys.objects AS o ON p.object_id = o.object_id
    JOIN sys.indexes AS i ON p.index_id = i.index_id AND i.object_id = p.object_id
```

```
and p.index_id=0
ORDER BY o.name, p.index_id;
```

# Demonstration 1: Index and stats views



# Demonstration 2: Clustered Indexes



**Purpose:**

Locate Clustered Indexes in your database

**Objective:**

Query the system views to find clustered indexes on your tables..

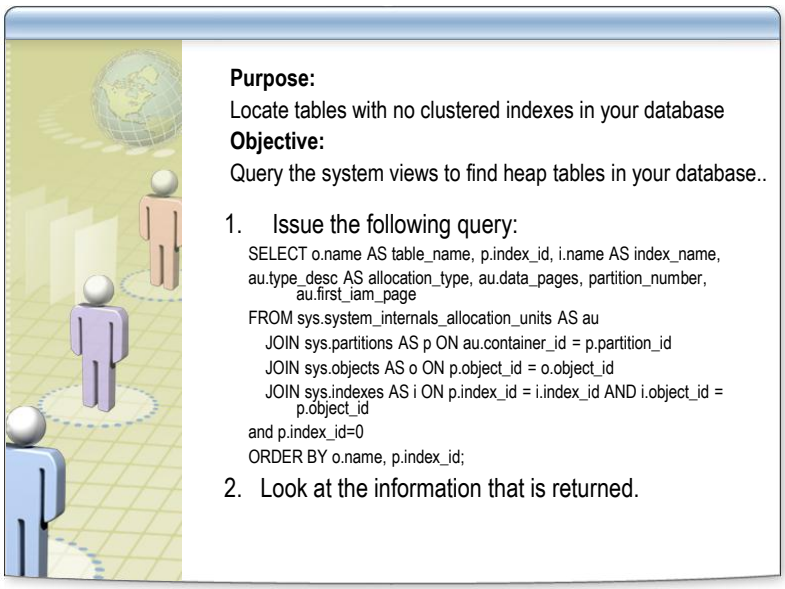
1. Issue the following query:

```
SELECT o.name AS table_name, p.index_id, i.name AS index_name,
au.type_desc AS allocation_type, au.data_pages, partition_number,
    au.root_page
FROM sys.system_internals_allocation_units AS au
   JOIN sys.partitions AS p ON au.container_id = p.partition_id
   JOIN sys.objects AS o ON p.object_id = o.object_id
   JOIN sys.indexes AS i ON p.index_id = i.index_id AND i.object_id =
       p.object_id
and p.index_id=1
ORDER BY o.name, p.index_id
```

2. Look at the information that is returned.

20

# Demonstration 3: Heap Tables



**Purpose:**

Locate tables with no clustered indexes in your database

**Objective:**

Query the system views to find heap tables in your database..

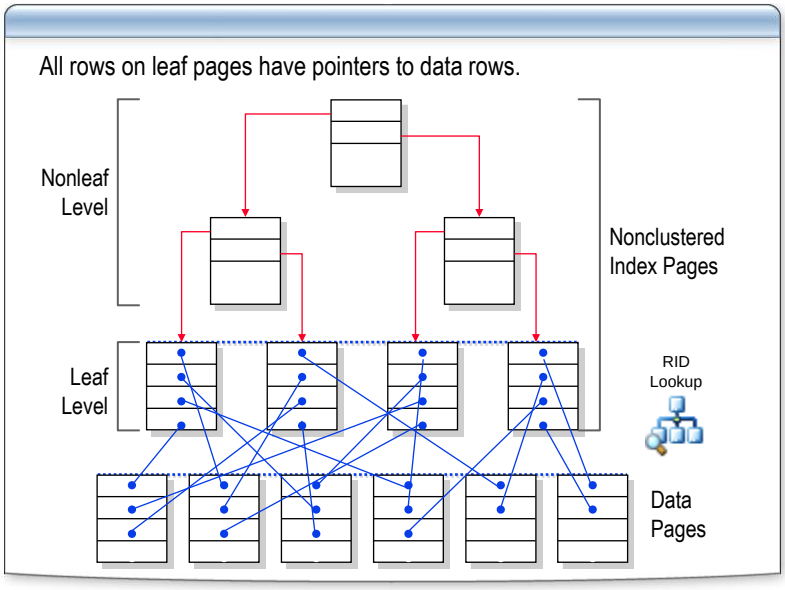
1. Issue the following query:

```
SELECT o.name AS table_name, p.index_id, i.name AS index_name,
au.type_desc AS allocation_type, au.data_pages, partition_number,
      au.first_iam_page
FROM sys.system_internals_allocation_units AS au
   JOIN sys.partitions AS p ON au.container_id = p.partition_id
   JOIN sys.objects AS o ON p.object_id = o.object_id
   JOIN sys.indexes AS i ON p.index_id = i.index_id AND i.object_id =
      p.object_id
and p.index_id=0
ORDER BY o.name, p.index_id;
```

2. Look at the information that is returned.

21

# Non-Clustered Indexes with a Heap




In a non-clustered index, the leaf level of the tree contains an RID that points to the location of the data row corresponding to the index key.

If the table is a heap, as shown in the illustration on the slide above, the RID is a pointer to the row. The pointer is built from the file ID, page number, and slot ID of the row on the page. The entire pointer is the RID.

When SQL Server uses a non-clustered index to search for a record, it traverses the B-tree until it gets to the indexed value for the record. Then, it uses the RID to go straight to the data in the data pages. The RID never needs to be updated, because when the data is stored in a heap, the location of the data never changes. If a row is updated and it no longer fits on the original page, SQL Server stores a page forwarded pointer to the row overflow page and slot where the column will be stored.

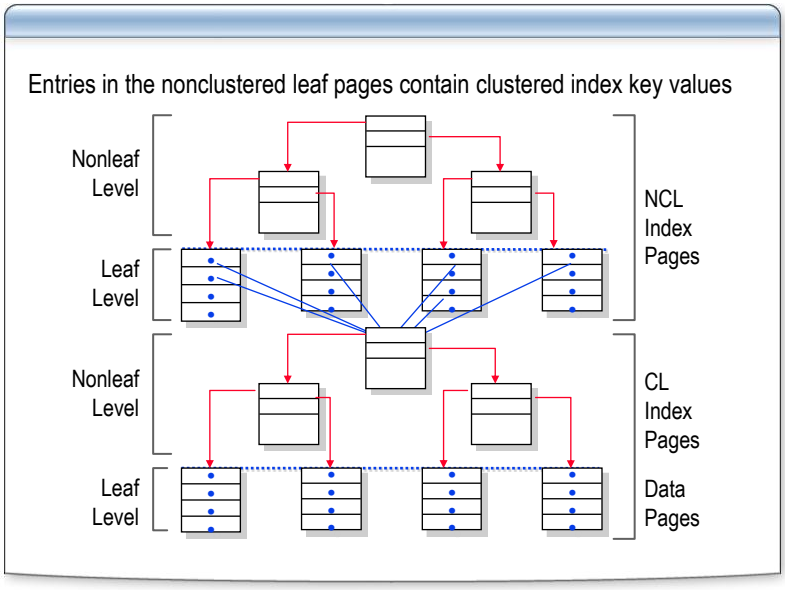
Non-clustered indexes are useful for fetching a limited number of rows with good selectivity from large SQL Server tables, based on a key value. The leaf level of the B-tree of index pages contains all the data from the columns that comprised that index. When a non-clustered index is used to retrieve information from a table, based on a key value, the index B-tree is traversed until a key match is found at the leaf level of the index. This index seek works the same as the clustered index seek described earlier.

If the requested columns for the query are not included in the index, an RID lookup is executed. This RID lookup may require a non-sequential I/O operation on the disk. It might even require the data to be read from another disk, if the table and its accompanying index B-tree(s) are large. If multiple RID lookups lead to the same 8-KB

data page, then there is less of an I/O performance penalty, because it is only necessary to read the page into data cache once. These RID lookups are the reason that non-clustered indexes are better suited for SQL queries that return only one or few rows from the table. Queries that require many rows to be returned where the columns are not covered by the non-clustered index are better served with either a clustered index scan or a table scan.

The **RID Lookup** operator is always accompanied by a NESTED LOOP JOIN Operator in the execution plan. This pair of operators replaces the Bookmark lookup from earlier versions of SQL Server.

# Non-Clustered Index with a Clustered Index




What happens when SQL Server searches a non-clustered index and there is a clustered index on the table? For example, the query asks for the first, middle, and last names of all people whose first name is Joe, and the employee table has a clustered index on last names and a non-clustered index on first names.

SQL Server traverses the non-clustered index until it gets to the record that has the indexed value of **Joe**. Since the clustered index keys appear in the non-clustered index, the last name value can also be read from the non-clustered index. If this was all that was queried, there would be no need to look up values in the base table. However; the middle name is not included in the non-clustered or clustered index keys, and we have no included columns. To retrieve the middle name, SQL Server traverses the clustered index.

This method seems like a lot of extra work to get a single value. The reason for using this method is that when the pointer is the clustered index key, there are fewer changes to the non-clustered index during inserts, updates, and deletes. This saves time on large tables.

Because the clustered index key appears in the non-clustered index, the size of the clustered index key contributes to the space required by each non-clustered index record. The larger the clustered index, the more data pages are required by the non-clustered index, thereby increasing the amount of I/O needed to traverse the index. Therefore, it is important to keep clustered index keys as narrow as possible.

The power of the clustered index is that a large number of columns can be returned as a range without any extra lookups being performed—all of the data appears on the leaf pages of the clustered index. If columns that do not appear in the index keys or included

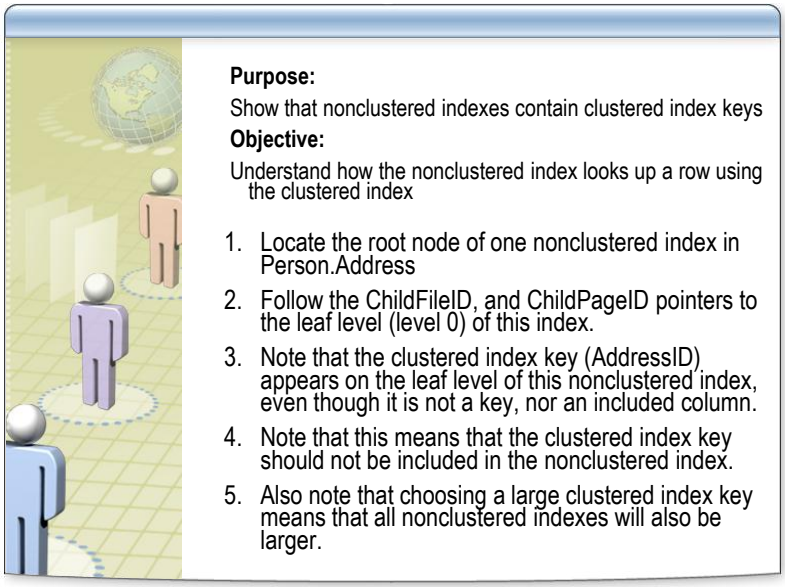
columns of a non-clustered index are requested, then additional work must be done in the form of either a RID lookup or a KEY lookup in order to retrieve the values that do not appear in the non-clustered index. If a large range is requested, then it will become less expensive to perform a table scan or clustered index scan than to perform the index seek with the RID or KEY lookup.

If a large range of rows is requested with only a few columns, then a non-clustered index can be created to cover the query and use fewer page reads than a clustered index for that query. In these cases, a covering non-clustered index is better for this query.

If a large number of columns is usually requested on a large range, then this is a situation where a clustered index may be more suitable.

By default, the Primary Key constraint will be created as a unique, clustered index unless non-clustered is specified or unless there is already a clustered index on the table. You should not just accept this as optimal. Refer to the discussion on choosing the right index type and evaluate carefully whether the primary key is the most optimal choice for a clustered index on a table.

# Demonstration 4: Nonclustered Index Structure (Optional)



**Purpose:**

Show that nonclustered indexes contain clustered index keys

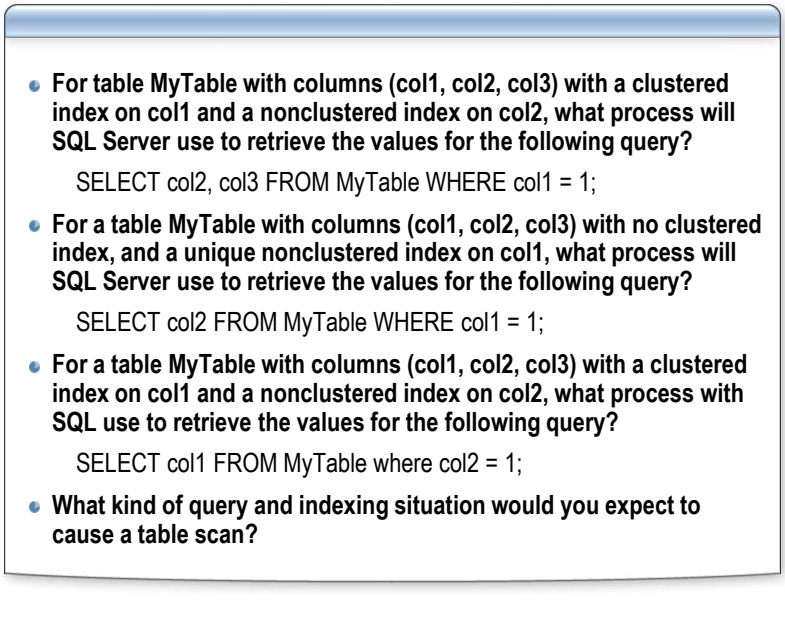
**Objective:**

Understand how the nonclustered index looks up a row using the clustered index

1. Locate the root node of one nonclustered index in Person.Address
2. Follow the ChildFileID, and ChildPageID pointers to the leaf level (level 0) of this index.
3. Note that the clustered index key (AddressID) appears on the leaf level of this nonclustered index, even though it is not a key, nor an included column.
4. Note that this means that the clustered index key should not be included in the nonclustered index.
5. Also note that choosing a large clustered index key means that all nonclustered indexes will also be larger.

24

# Section 3 Review




- For a table MyTable with columns (col1, col2, col3) with a clustered index on col1 and a non-clustered index on col2, what process will SQL use to retrieve the values for the following query?

```
SELECT col2, col3 FROM MyTable WHERE col1 = 1;
```

- For a table MyTable with columns (col1, col2, col3) with no clustered index, and a unique non-clustered index on col1, what process will SQL use to retrieve the values for the following query?

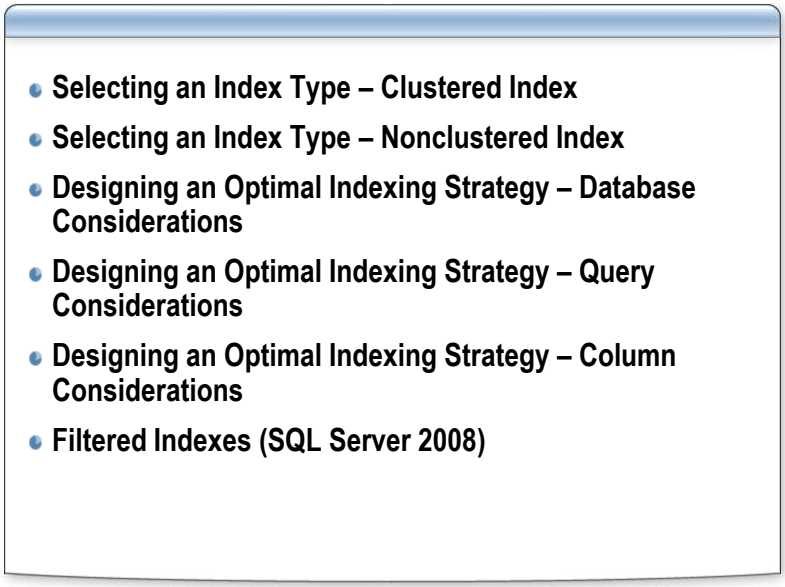
```
SELECT col2 FROM MyTable WHERE col1 = 1;
```

- For a table MyTable with columns (col1, col2, col3) with a clustered index on col1 and a non-clustered index on col2, what process will SQL use to retrieve the values for the following query?

```
SELECT col1 FROM MyTable where col2 = 1;
```

- What kind of query and indexing situation would you expect to cause a table scan?

# Section 4: Developing an Index Strategy




## Introduction

Every design optimization decision in a database comes at a price. There is a tradeoff for every change made. Because of this, there is no one solution that will work optimally for every situation. The process of optimization involves understanding these tradeoffs and making informed decisions on whether or not the gain is worth the cost. In the end, testing will be the only way to determine if the optimization has given the desired net gain. An understanding of a few principles can help you make good optimization decisions.

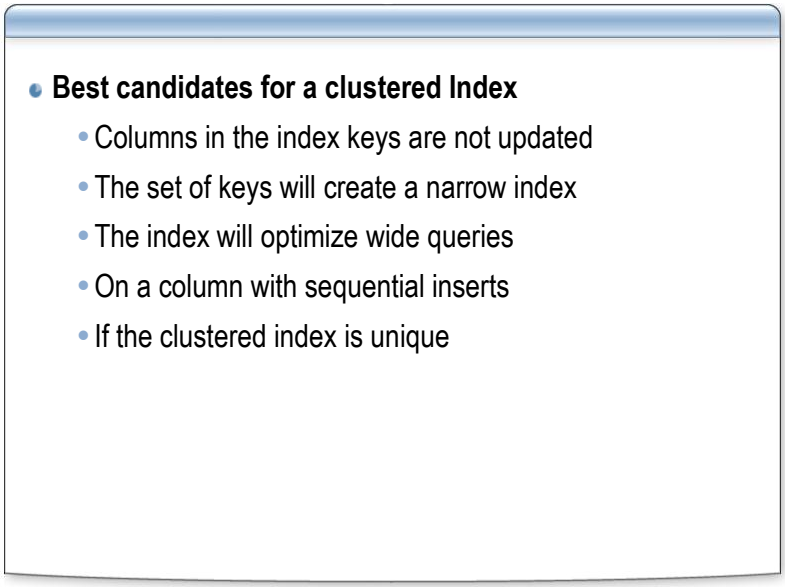
This section covers some considerations from the database, query, and column perspective and explains a few ways that you can use indexing to help SQL access the data optimally.

## Objectives

After completing this section, you will be able to:

- Select an index type for a given set of requirements.
- Design an indexing strategy that is optimal for the database load.

# Selecting an Index Type—Clustered Index




A clustered index should be considered for most tables. A clustered index optimizes inserts by allowing the insertion process to navigate the B-tree to the leaf page where the insert must take place, rather than scanning the PFS pages and IAM pages, which is required for inserting a new row into a heap table. Clustered indexes are preferred in queries where an RID lookup or KEY lookup may otherwise need to be performed to return all columns.

Because the data physically resides on the leaf pages of the clustered index, there can only be one clustered index on a table. Because of its importance to all other indexes, and its importance to queries, there are a few things that you should consider when deciding which index is the best candidate to be a clustered index. The following sections describe these considerations.

## Columns in the index keys are not updated

Remember that the data resides on the leaf pages of the clustered index. This means that the clustered index will probably have more data for each row on its leaf page than any other index. Because it is part of the index, the values of the index keys determine on which page the data row will be located. If the clustered index key is updated, it is likely that the entire data row must be moved from its current location to a new page based on the new values of the clustered index keys. Additionally, since the clustered index key is stored in all non-clustered indexes, updating the clustered index key means that all nonclustered indexes will need to be updated.

Because this data movement and extra updating can create a big performance penalty on updates, columns that are frequently updated are probably not the best candidates for clustered index keys. You should create clustered indexes on non-updated or seldom updated columns.

### The set of keys will create a narrow index

Remember that the clustered index keys appear in all non-clustered indexes on a table. If the clustered index is created on a single integer column, then each entry at the intermediate levels of the clustered index is only 4 bytes, and 4 bytes are added to each entry in every non-clustered index as well. But if the clustered index is created on two CHAR(100) columns, then not only is the size of the intermediate levels of the clustered index increased, but an additional 200 bytes is added to every entry in every non-clustered index on this table as well.

In effect, such a wide clustered index decreases the efficiency of every non-clustered index on the table by making a larger number of page reads necessary every time that the non-clustered index is used for a seek or a scan. The clustered index and all the non-clustered indexes on that table also require additional storage. To avoid these penalties, you should consider using narrow clustered index keys.

### The index will optimize wide queries

Consider the following situation:

```
USE AdventureworksPTO;
go;

-- hit CTRL-L on the next statement. Notice that it uses an index seek
-- and because of the large number of columns
-- it must do a key lookup to get the rest of the data

SELECT SalesOrderID, RevisionNumber, OrderDate
,      DueDate, ShipDate, Status, OnlineOrderFlag, SalesOrderNumber
,      PurchaseOrderNumber, AccountNumber, CustomerID
,      SalesPersonID, TerritoryID, BillToAddressID, ShipToAddressID
,      ShipMethodID, CreditCardID, CreditCardApprovalCode
FROM Sales.SalesOrderHeader WHERE SalesOrderNumber = 'SO58658'
```

When you press CTRL+L with the query shown above highlighted, you should see an index seek and a key lookup. The non-clustered index used is on the SalesOrderNumber, but it is a narrow index, and therefore, it needs to look up the rest of the values in the SELECT list from the clustered index. This key lookup is very expensive, so when there are many rows, SQL will no longer use this method. See what happens if you want about 10 percent of the values from this table:

```
-- but what if we ask for about 3,000 of the 30,000 rows?
-- what happens with the query plan. Hit ctrl-l to see:

SELECT SalesOrderID, RevisionNumber, OrderDate
,      DueDate, ShipDate, Status, OnlineOrderFlag, SalesOrderNumber
,      PurchaseOrderNumber, AccountNumber, CustomerID
```

```
,       SalesPersonID, TerritoryID, BillToAddressID, ShipToAddressID
,       ShipMethodID, CreditCardID, CreditCardApprovalCode
FROM Sales.SalesOrderHeader
WHERE SalesOrderNumber BETWEEN 'SO43659' AND 'SO46658';
```

When you press CTRL+L with the query shown above highlighted, and look at the query plan, you should get a clustered index scan. In this scenario, SQL has determined that it is cheaper to scan the clustered index than to perform 3000 key lookups.

To get rid of that possibility, you could create a non-clustered index and include all of the columns in the SELECT list, but this is a very wide query. This covering index would also be very wide.

If you change the filtering from SalesOrderNumber to SalesOrderID, you will see that SQL handles wide queries differently if it can use the clustered index. Highlight each of the following queries and press CTRL+1 to view the difference:

```
-- get one row using the clustered index. There is no lookup because
-- the data is on the leaf pages of the clustered index:

SELECT SalesOrderID, RevisionNumber, OrderDate
,       DueDate, ShipDate, Status, OnlineOrderFlag, SalesOrderNumber
,       PurchaseOrderNumber, AccountNumber, CustomerID, ContactID
,       SalesPersonID, TerritoryID, BillToAddressID, ShipToAddressID
,       ShipMethodID, CreditCardID, CreditCardApprovalCode
from Sales.SalesOrderHeader WHERE SalesOrderID = 43659

-- if we ask for a range of 15,000 rows with this large number of columns
-- using the clustered index we get no lookup, and no degradation
-- to a clustered index scan. This is because
-- all the data is physically on the leaf pages of the clustered index

SELECT SalesOrderID, RevisionNumber, OrderDate
,       DueDate, ShipDate, Status, OnlineOrderFlag, SalesOrderNumber
,       PurchaseOrderNumber, AccountNumber, CustomerID, ContactID
,       SalesPersonID, TerritoryID, BillToAddressID, ShipToAddressID
,       ShipMethodID, CreditCardID, CreditCardApprovalCode
from Sales.SalesOrderHeader WHERE SalesOrderID BETWEEN 43659 AND 58658
```

A non-clustered index to cover such a wide query and prevent the degradation to a clustered index or table scan would be a very wide index, and would almost be a second copy of the table. However, the data already exists in the leaf pages of the clustered index; therefore, a narrow clustered index on the appropriate column will return the results as efficiently as possible without requiring a second copy of several columns. You should consider such wide queries when choosing the clustered index, and choose a clustered index that will optimize as many such queries as possible.

## Clustered index is best on a column with sequential inserts

As mentioned earlier, a clustered index is usually the largest index on a table because the data appears in the leaf nodes of the index. With such a large amount of data in the leaf pages, the clustered index will likely experience the most page splits because more data is

inserted in a clustered index than any other index. Page splits are performed in two different ways depending on the pattern of insertion.

If data is inserted anywhere other than beyond the current range of the index, then in the event of a page split, a new page will be allocated, about half of the data from the existing page will be moved to the new page, and the new row that caused the page split will be placed into the appropriate place in either the original page or the new page. The movement of half of the data from one page to the new page is a relatively expensive operation, and it leaves you with two pages that are half full, but both of which occupy 8 KB of buffer and storage.

If data is inserted in sequence, then in the event of a page split, no data is copied to the new page. Instead, the new page is allocated, and the new value is placed into this new page. Because no data is moved, this is much less expensive than non-sequential page splits. Because the data is inserted sequentially, new inserts will go only to this new page until it is full, and so you do not have large numbers of half-full pages due to page splits. This optimizes both the page split and the use of memory and storage.

It is the clustered index that will experience the largest number of page splits and contains the most data. Therefore, creating a clustered index on a column with sequential inserts optimizes inserts, and minimizes the number of page reads for queries that use the index. Columns with sequentially inserted data make good candidates for a clustered index.

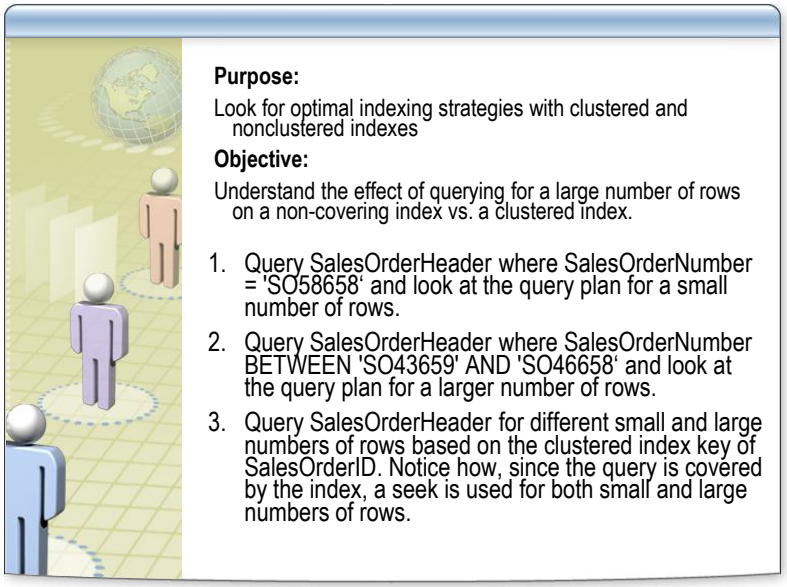
### Clustered index is best as a unique index

SQL always creates a clustered index as a unique index. If the clustered index is not created as a unique index, then SQL adds a 4-byte Uniqueifier column to every level in the clustered index. This is additional storage in the clustered index itself, but you should remember that the clustered index keys appear in all the non-clustered indexes. If the clustered index is not created as unique, then the extra 4-byte uniqueifier appears in the non-clustered indexes as well. However, the additional storage requirement goes even farther than this.

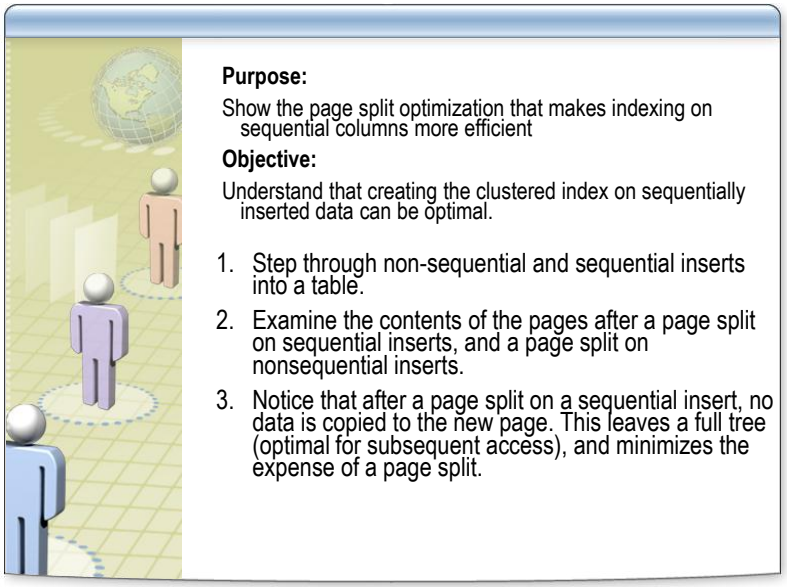
If the clustered index is unique, then the clustered index keys only appear on the leaf level of the non-clustered indexes. If both the clustered and non-clustered indexes are not unique, then the clustered index keys will appear at the root and intermediate levels of the non-clustered index as well.

The additional values added to the non-clustered indexes require additional storage, and will cause more page reads when the non-clustered indexes are used. To prevent these penalties, you should create the clustered index as a unique index whenever possible.

# Demonstration 5: Access Methods

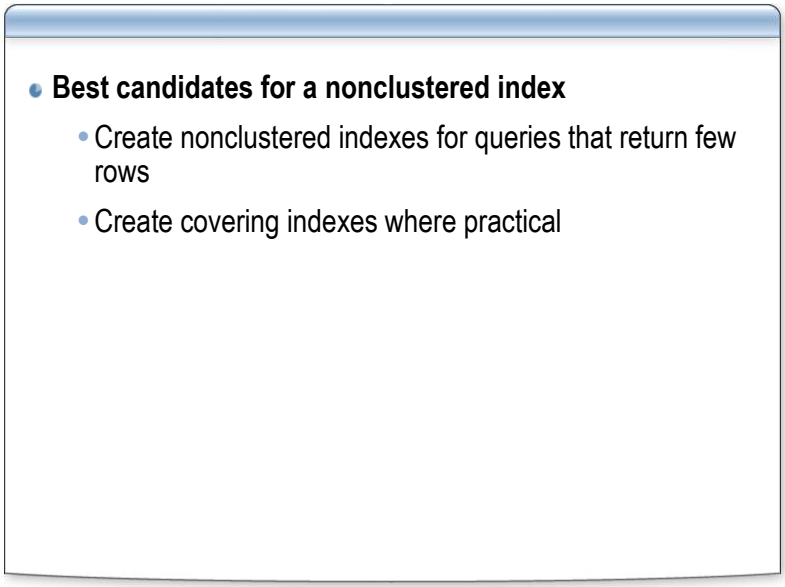


# Demonstration 6: Optimization for Sequential Inserts (Optional)




29

# Selecting an Index Type—Non-Clustered Index




A table is usually queried in multiple ways; it is very seldom that one table is queried in only one way. Each table will be queried in different manners for different needs. Therefore, it will likely need more than one index to efficiently satisfy all queries against it. Because you can have only one clustered index, all other indexes on a table must be non-clustered.

## Create non-clustered indexes for queries that return few rows

Consider the following example from the previous topic on clustered indexes:

```
USE AdventureworksPTO;
go;

-- hit ctrl-l on the next statement. Notice that it uses an index seek
-- and because of the large number of columns
-- it must do a key lookup to get the rest of the data

SELECT SalesOrderID, RevisionNumber, OrderDate
,      DueDate, ShipDate, Status, OnlineOrderFlag, SalesOrderNumber
,      PurchaseOrderNumber, AccountNumber, CustomerID
,      SalesPersonID, TerritoryID, BillToAddressID, ShipToAddressID
,      ShipMethodID, CreditCardID, CreditCardApprovalCode
from Sales.SalesOrderHeader WHERE SalesOrderNumber = 'SO58658'
```

If you examine the query plan, you can see that this query uses the non-clustered index on SalesOrderNumber to find the one row that it needs, and then uses a key lookup to find all of the values in the SELECT list.

This lookup will degrade into a clustered index scan if you query for large numbers of rows. To avoid the clustered index scan, you can create a non-clustered index that includes all columns in the SELECT list, but this will be a very wide index and may be more expensive to maintain than the benefit you will receive from it.

If you query for a large number of rows by using the current non-clustered index, then your query plan will change to use a clustered index scan. But if you use this method to receive only one or two rows, then this non-clustered index is likely all that you need. It will perform the key lookup, but on a few rows, that is not prohibitively expensive. In this case, a narrow non-clustered index may serve the purpose nicely.

## Create covering indexes where practical

A covering index is one that contains all the data needed to resolve a query without needing to refer to the base table. Consider a table tab1 with columns (col1, col2, col3, and col4). If you have a non-clustered index on (col1, col2) and you have a query "SELECT col2 FROM tab1 WHERE col1 = 'value'", then the non-clustered index covers the query. SQL will use a seek to locate the rows where col1 = 'value', and then, because it only needs col2, it will read col2 directly from the leaf pages of the index to return the values for the query. It will never need to do a key lookup or a RID lookup to find additional values.

Beginning with SQL Server 2005, SQL offered the option to **INCLUDE** columns in an index. An included column is not an index key, so it does not appear at the root or intermediate levels of the index, but rather appears only on the leaf pages of the non-clustered index. This allows you to build covering indexes without bloating the intermediate levels of the index. So on the table tab1 above, if you create an index as follows:

```
CREATE INDEX ncl_tab1 ON tab1 (col2) INCLUDE (col1, col3);
```

Then that index will also cover the query "SELECT col1, col3 FROM tab1 WHERE col2 = 'value'".

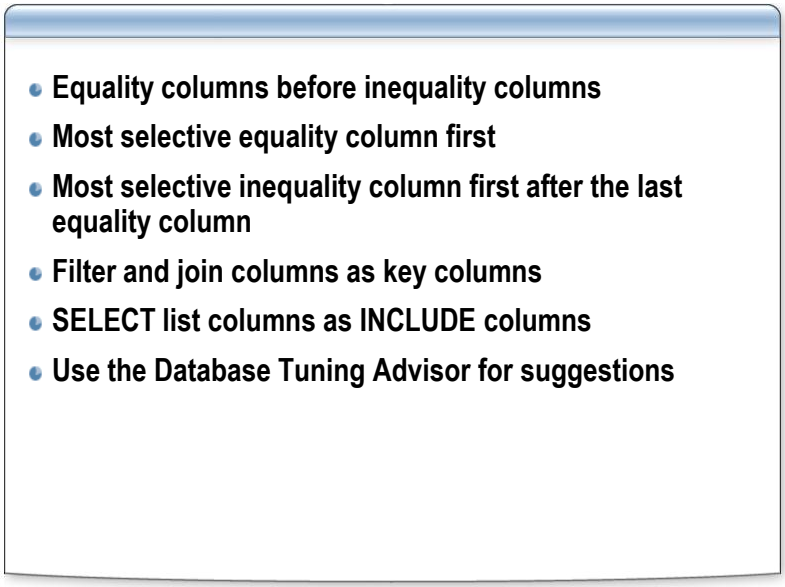
Consider the following example of saving space. You have a 100-million row index that has a key length of 900 bytes. Only the first two integer keys are required as the index keys. With the 900-byte key, eight rows can fit on each database page (this means that the fanout is 8). This means that there will be 12.5 million pages at the leaf level, 1.6 million pages at the next level up in the B-tree, and so on, resulting in a total of 12,500,000 + 1,562,500 + 195,313 + 24,415 + 3,052 + 382 + 48 + 6 + 1 = 14.3 million pages (including 1.8 million pages to store the upper levels of the B-tree).

If you create an index on only the first two integers, and include the other columns, then the key size shrinks to 8 bytes. In addition, with the row overhead, you can get the row length in the upper levels of the B-tree down to 15 bytes (giving a fanout of approximately 537). Note that the fanout at the leaf level is still going to be 8, because the amount of data stored in each row at the leaf level is the same. So, this means that there will be 12.5 million pages at the leaf level, 23,278 pages at the next level up, and so

on, giving a total of 1,250,0000 + 23,278 + 44 + 1 = 12.52 million pages (including 23,323 pages to store the upper levels of the B-tree). When you compare this to the full-size 900-byte key, this is a 12 percent, or 1.8 million pages, or 13.6 GB saving.

Covering indexes allow you to query for individual rows or ranges of rows much more efficiently than you could with just a clustered index. Unlike the clustered index, the non-clustered index will contain only the non-clustered index keys, the included columns, and the clustered index keys at its leaf level. For narrow queries, you can create relatively narrow covering indexes, return a range of rows with fewer page reads, and therefore, incur a lower expense than with a clustered index.

# Designing an Optimal Indexing Strategy—Column Considerations




After you have selected the index type, you must design the index itself for most efficient data access. When multiple queries require different variations of the same indexes, this can be somewhat tricky. In general, you can apply a few principles to arrive at an optimal index for a query, but you must consider the tradeoff with other queries against the table. In practice, you must find a tradeoff for the number of indexes, and the best use of those indexes across the load. Here are a few principles to follow when designing an optimal indexing strategy.

## Equality columns before inequality columns

Consider a table that contains a bit column named bit1, and an integer column named int1. If you issue the following query:

```
SELECT col3 FROM myTable where bit1 = 1 AND int1 BETWEEN 1 AND 10;
```

Then, bit1 is the equality column because it is evaluated only where it is equal to a value, and int1 is an inequality column because it is evaluated for a range of values. Look now at the possibilities for index keys, and which will be more optimal.

Int1 is obviously more selective than bit1 because only two possible values exist for a bit column. But if the following combination exists for col1, int1, then it is more efficient to order the data by the bit column first:

**Sort by int1, bit1**

| int1 | bit1 |
|---|---|
| 1 | 0 |
| 1 | 1 |
| 2 | 0 |
| 2 | 1 |
| 3 | 0 |
| 3 | 1 |
| 4 | 0 |
| 4 | 1 |
| 5 | 0 |
| 5 | 1 |
| 6 | 0 |
| 6 | 1 |
| 7 | 0 |
| 7 | 1 |
| 8 | 0 |
| 8 | 1 |
| 9 | 0 |
| 9 | 1 |
| 10 | 0 |
| 10 | 1 |

**Sort by bit1, int1**

| int1 | bit1 |
|---|---|
| 1 | 0 |
| 2 | 0 |
| 3 | 0 |
| 4 | 0 |
| 5 | 0 |
| 6 | 0 |
| 7 | 0 |
| 8 | 0 |
| 9 | 0 |
| 10 | 0 |
| 1 | 1 |
| 2 | 1 |
| 3 | 1 |
| 4 | 1 |
| 5 | 1 |
| 6 | 1 |
| 7 | 1 |
| 8 | 1 |
| 9 | 1 |
| 10 | 1 |

To resolve the query, you navigate the B-tree to the first value that you are querying, then scan page by page through the leaf level until you reach the last value that you are querying. If you sort by the more selective int1 first, then you must read every value from 1, 1 through 10, 1. These are 19 rows that you must evaluate to retrieve the 10 values. If you sort by the equality column bit1 first, then you arrive at the leaf level at the row containing 1, 1. You must then only read 10 values to retrieve all 10 values that you are querying.

This is a very small dataset used only for illustration, but the principle applies to the larger datasets that you will likely be using in SQL as well. This is counter to many database axioms, but most times, it is more important to have the equality column first than to have the most selective column first simply because putting the equality column first prevents the unwanted equality values that fall within the inequality range from appearing in the range read.

## Most selective equality column first

After the equality and inequality columns have been separated and prioritized with equality columns first, then you should put the most selective equality column first.

Putting the most selective equality column first allows for the number of reads to be minimized, because the first column limits the range read first.

### Most selective inequality columns after the last equality column

Consider the following query:

```
SELECT * FROM tab1 WHERE col1 = 2 AND col2 = 3 AND col4 > 10 AND col5 > 15
```

You have two equality columns, col1 and col2, and two inequality columns, col4 and col5. If you determine col1 to be more selective than col2, and col5 to be more selective than col4, then to minimize the number of reads, your best choice on indexes would be on (col1, col2, col5, col4). This way, the most selective inequality column comes after the equality columns, and before the less selective inequality columns.

### Filter and JOIN columns as key columns

The index key columns appear in the root and intermediate levels of the B-tree and are used to navigate to the correct first page in the root level. All the columns in the JOIN or WHERE clauses play into this navigation, so they should appear as key columns in the index.

### SELECT columns as INCLUDED columns

If you want to create a covering index so that no RID or KEY lookups are performed, then you must find a place for the additional columns in the SELECT list in your index.

Consider a table with a clustered index on col1. If you run the following query on this table:

```
SELECT col1, col2, col3 FROM myTable WHERE col4 = 10 AND col5 > 20;
```

Then, you could create the following index to cover the query:

```
CREATE INDEX ncl_myTable_1 ON myTable(col4, col5) INCLUDE (col2, col3);
```

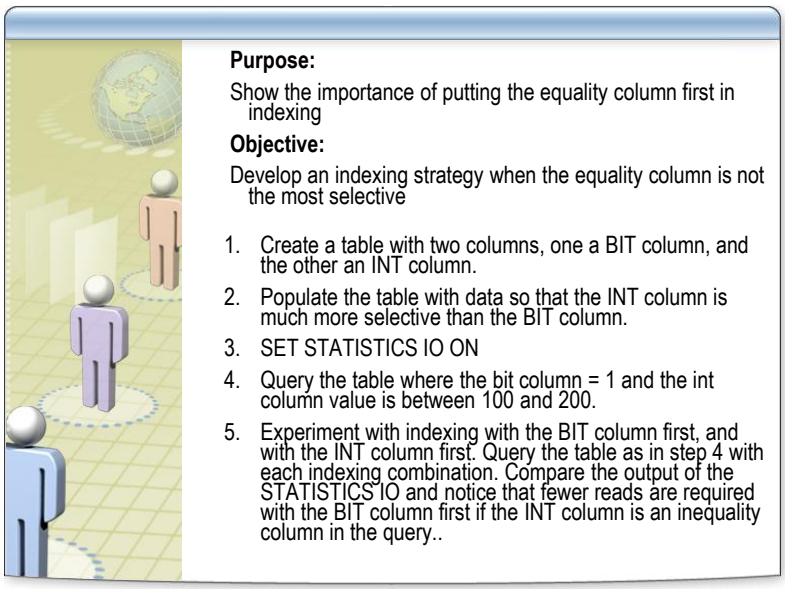
So, you have created the index with the equality column first, the filter columns as keys, and the other columns that appear in the SELECT list as INCLUDE columns.

Why was col1 not included? Remember that you have a clustered index on col1. Col1 already appears in the non-clustered index without requiring you to include it. The index covers the query without the need to specify col1.

### Use the Database Engine Tuning Advisor for suggestions

Often, you can use the Database Engine Tuning Advisor (DTA) for faster analysis of a load, and good suggestions for indexes to optimize a load. You should use the DTA for further suggestions on indexing.

# Demonstration 7: Equality Columns First



**Purpose:**

Show the importance of putting the equality column first in indexing

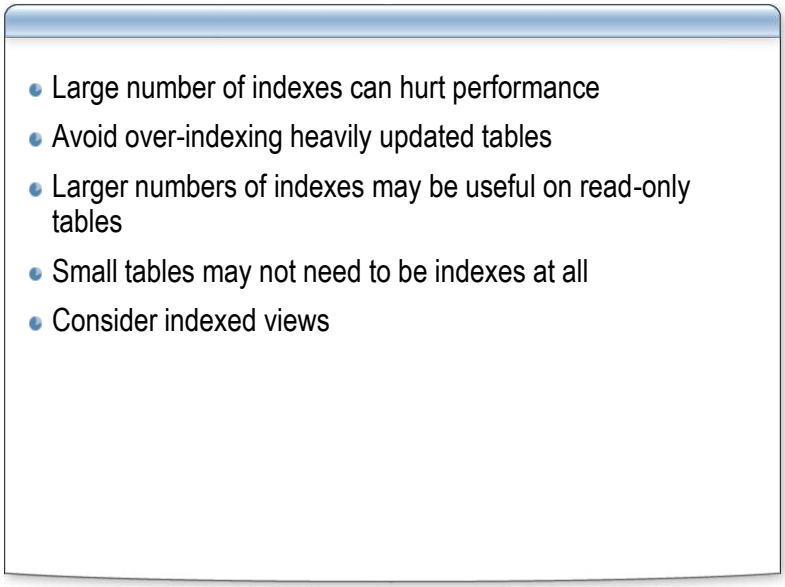
**Objective:**

Develop an indexing strategy when the equality column is not the most selective

1. Create a table with two columns, one a BIT column, and the other an INT column.
2. Populate the table with data so that the INT column is much more selective than the BIT column.
3. SET STATISTICS IO ON
4. Query the table where the bit column = 1 and the int column value is between 100 and 200.
5. Experiment with indexing with the BIT column first, and with the INT column first. Query the table as in step 4 with each indexing combination. Compare the output of the STATISTICS IO and notice that fewer reads are required with the BIT column first if the INT column is an inequality column in the query..

32

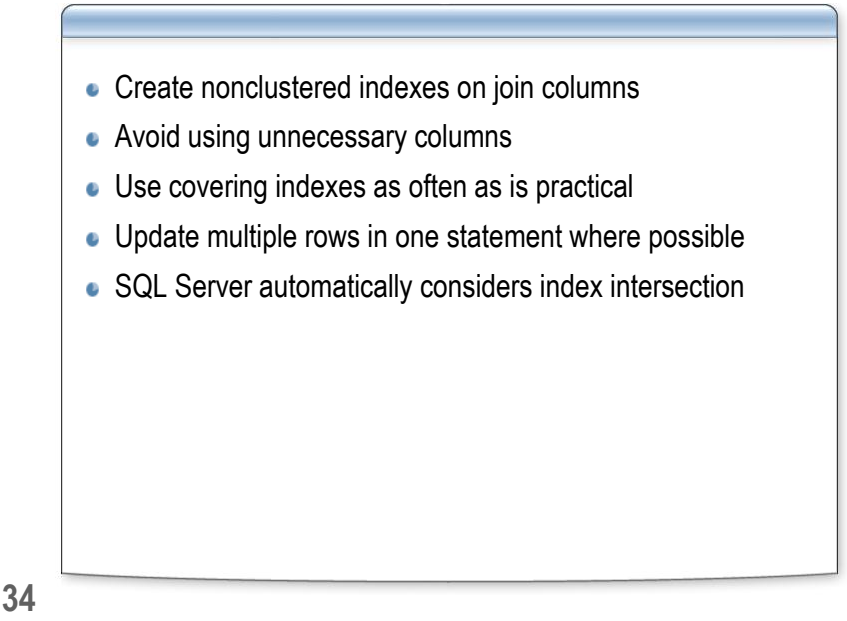
# Designing an Optimal Indexing Strategy—Database Considerations




When designing an index, you should consider the following database guidelines:

- Large numbers of indexes on a table affect the performance of INSERT, UPDATE, and DELETE statements because all indexes must be adjusted as data in the table changes. Indexes can help UPDATE or DELETE statements with WHERE clauses by efficiently locating the rows to be updated or deleted. You must evaluate the tradeoff carefully.
- Avoid over-indexing heavily updated tables and keep indexes narrow, that is, with as few columns as possible.
- Larger number of indexes may be used with minimal penalty on tables with low update requirements but large volumes of data. Larger number of indexes can help the performance of larger varieties of queries that do not modify data.
- Indexing small tables may not be optimal because it can take the query optimizer longer to evaluate the usefulness of an index than to perform a simple table scan. It may also take longer to navigate a small index than to simply perform a table scan. Therefore, indexes on small tables may not be used, but still incur maintenance overhead as the data in the table changes.
- Indexes on views can provide significant performance gains when the view contains aggregations, table joins, or a combination of aggregations and joins. The view does not need to be explicitly referenced in the query for the query optimizer to use it.

# Designing an Optimal Indexing Strategy – Query Considerations

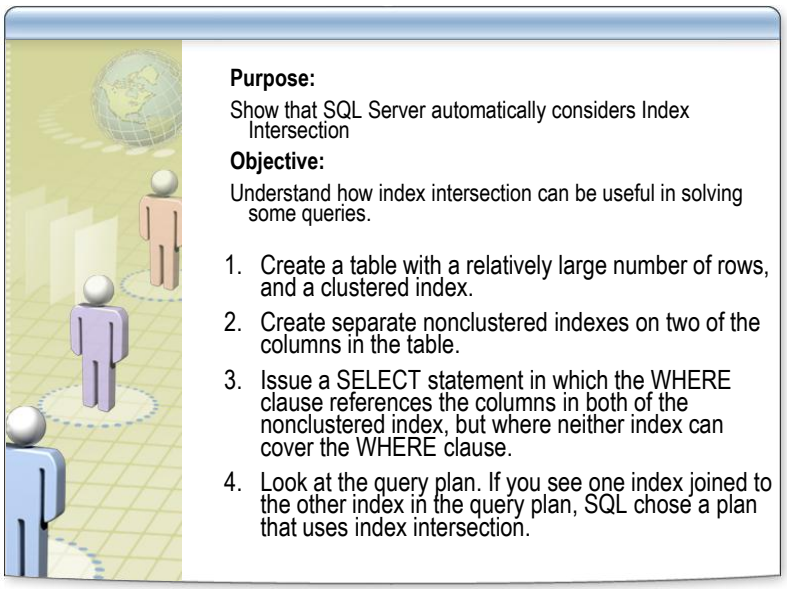



When designing an index, consider the following query guidelines:

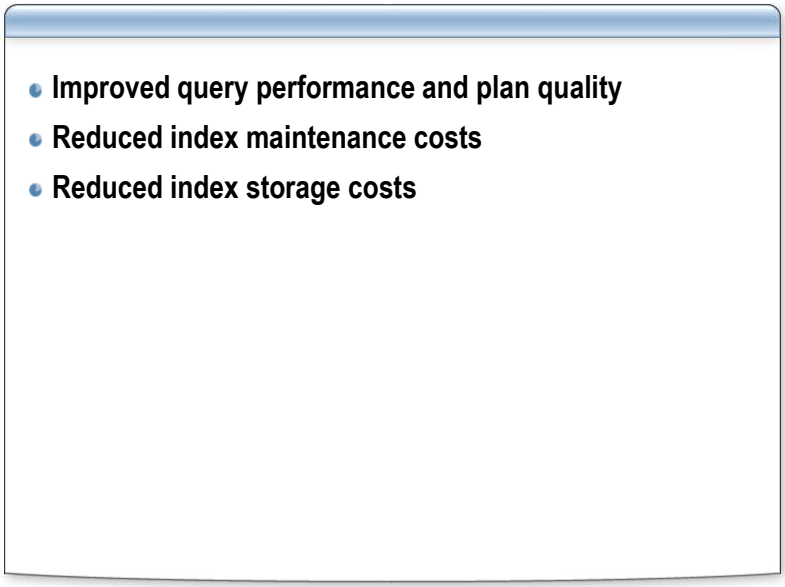
- Create non-clustered indexes on all columns that are frequently used in predicates and in join conditions in queries.
- Avoid adding unnecessary columns. Adding too many index columns can adversely affect disk space and index maintenance performance.
- Covering indexes can improve query performance because all the data needed to meet the requirements of the query exists within the index itself. This means that only the index pages, and not the data pages, of the table or clustered index, are needed to retrieve the requested data; therefore, using covering indexes reduces the overall disk I/O.
- Write queries that insert or modify as many rows as possible in a single statement, instead of using multiple queries to update the same rows. By using only one statement, you can exploit optimized index maintenance.

Do not use multiple aliases for a single table in the same query to simulate index intersection. This is no longer necessary, because SQL Server automatically considers index intersection and can use multiple indexes on the same table in the same query.

# Demonstration 8: Index Intersection (Optional)



# Filtered Indexes (SQL Server 2008)




A filtered index is an optimized non-clustered index, especially suited to cover queries that select from a well-defined subset of data. It uses a filter predicate to index a portion of rows in the table. A well-designed filtered index can improve query performance, reduce index maintenance costs, and reduce index storage costs compared with full-table indexes.

Filtered indexes can provide the following advantages over full-table indexes:

- **Improved query performance and plan quality**

  A well-designed filtered index improves query performance and execution plan quality because it is smaller than a full-table non-clustered index and has filtered statistics. The filtered statistics are more accurate than full-table statistics because they cover only the rows in the filtered index.

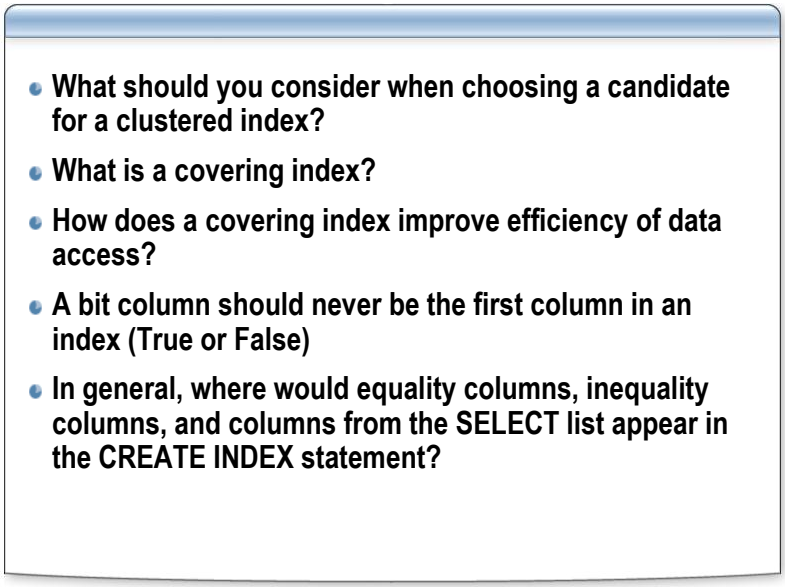
- **Reduced index maintenance costs**

  An index is maintained only when data manipulation language (DML) statements affect the data in the index. A filtered index reduces index maintenance costs compared with a full-table non-clustered index because it is smaller and is only maintained when the data in the index is affected. It is possible to have a large number of filtered indexes, especially when they contain data that is affected infrequently. Similarly, if a filtered index contains only the frequently affected data, the smaller size of the index reduces the cost of updating the statistics.

- **Reduced index storage costs**

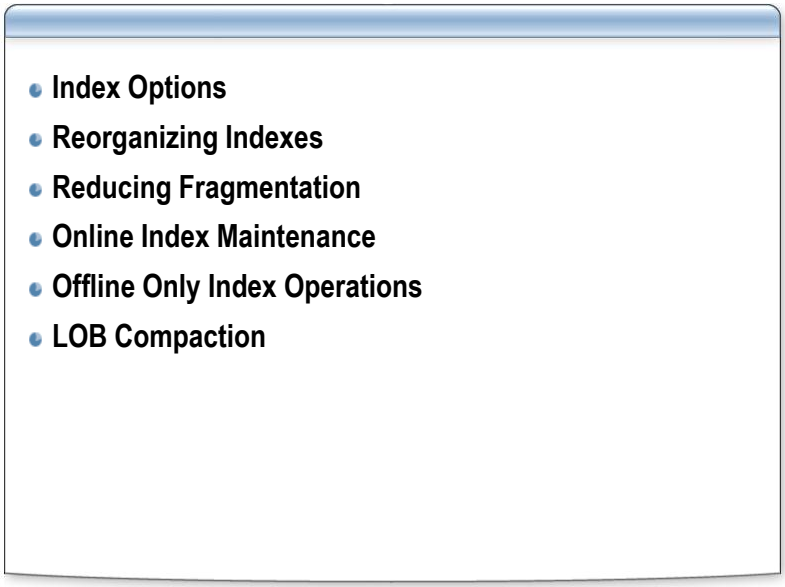
Creating a filtered index can reduce disk storage for non-clustered indexes when a full-table index is not necessary. You can replace a full-table non-clustered index with multiple filtered indexes without significantly increasing the storage requirements.

## Section 4 Review




- What should you consider when choosing a candidate for a clustered index?
- What is a covering index?
- How does a covering index improve the efficiency of data access?
- A bit column should never be the first column in an index. (True/False)
- In general, where would equality columns, inequality columns, and columns from the SELECT list appear in the CREATE INDEX statement?

# Section 5: Optimizing and Maintaining Indexes




## Introduction

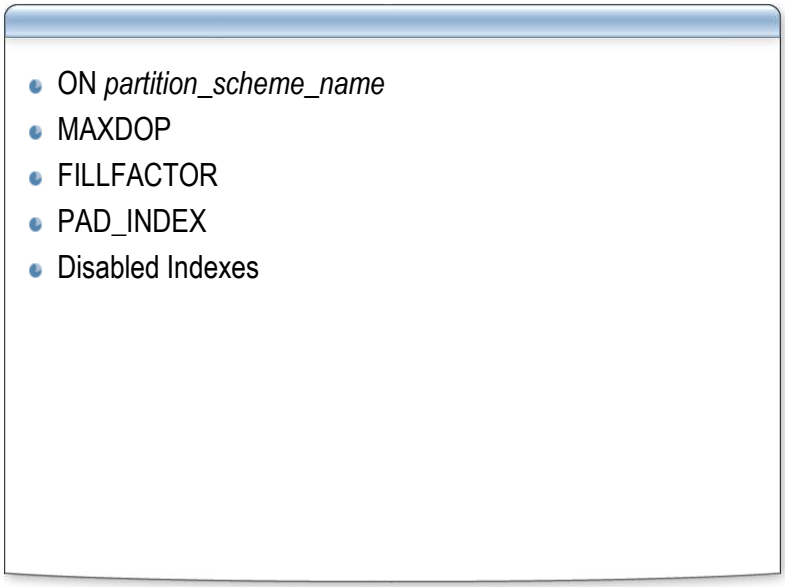
So far, you have learned about index selection and design. You can further optimize indexes with some new features that became available in SQL Server 2005 or 2008. This section describes index optimization and maintenance.

## Objectives

After completing this section, you will be able to:

- Describe the various options available for configuring and maintaining indexes.
- Detect index fragmentation in a table or heap.
- Reduce index fragmentation by recreating, reorganizing, or rebuilding the index.
- Identify the online indexing operations and maintenance activities that you can perform on an underlying table or index.
- Identify the indexing operations and maintenance activities that you can only perform offline.
- Explain why you should compact LOB data when reorganizing indexes.

# Index Options




Beginning with SQL Server 2005, several new indexing options have become available. These are designed to allow for more flexibility in using available resources for maintaining indexes, and for greater availability by allowing for online maintenance. These options are specified in the OPTION clause of the CREATE INDEX or ALTER INDEX statements. The full syntax with options is shown below:

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX index_name
    ON <object> ( column [ ASC | DESC ] [ ,...n ] )
    [ INCLUDE ( column_name [ ,...n ] ) ]
    [ WHERE <filter_predicate> ]
    [ WITH ( <relational_index_option> [ ,...n ] ) ]
    [ ON { partition_scheme_name ( column_name )
         | filegroup_name
         | default
         }
    ]
    [ FILESTREAM_ON { filestream_filegroup_name | partition_scheme_name |
"NULL" } ]

[ ; ]

<object> ::=
{
    [ database_name. [ schema_name ] . | schema_name. ]
        table_or_view_name
}

<relational_index_option> ::=
{
```

```
    PAD_INDEX = { ON | OFF }
  | FILLFACTOR = fillfactor
  | SORT_IN_TEMPDB = { ON | OFF }
  | IGNORE_DUP_KEY = { ON | OFF }
  | STATISTICS_NORECOMPUTE = { ON | OFF }
  | DROP_EXISTING = { ON | OFF }
  | ONLINE = { ON | OFF }
  | ALLOW_ROW_LOCKS = { ON | OFF }
  | ALLOW_PAGE_LOCKS = { ON | OFF }
  | MAXDOP = max_degree_of_parallelism
  | DATA_COMPRESSION = { NONE | ROW | PAGE}
     [ ON PARTITIONS ( { <partition_number_expression> | <range> }
     [ , ...n ] ) ]
}
```

**Note**: Filtered indexes using the WHERE clause are only available in SQL Server 2008.

### ON partition_scheme_name

The **ON *partition_scheme_name*** specifies the partition scheme that defines the filegroups onto which the partitions of a partitioned index will be mapped.

**Note**: When you partition a non-unique, clustered index, the database engine, by default, adds the partitioning column to the list of clustered index keys, if it is not already specified. When partitioning a non-unique, non-clustered index, the database engine adds the partitioning column as a nonkey (included) column of the index, if it is not already specified.

If partition_scheme_name or filegroup is not specified and the table is partitioned, the index is placed in the same partition scheme by using the same partitioning column as the underlying table.

### MAXDOP

The MAXDOP clause overrides max degree of parallelism configuration option for the duration of the CREATE INDEX statement.

### FILLFACTOR

The FILLFACTOR command determines the amount of data that the data page is loaded with, when the index is built or rebuilt. A FILLFACTOR of 100 attempts to fill each page completely. A FILLFACTOR of 10 only attempts to fill 10 percent of the capacity in each data page.

You should consider the maintenance window for rebuilding indexes when setting a value for FILLFACTOR:

- Lower fill factors enable more time before page splitting starts and therefore, more time before the index must be rebuilt to eliminate fragmentation.

- Higher fill factors decrease the number of I/Os necessary to cache data, which improves performance.

### PAD_INDEX

The PAD_INDEX option specifies the amount of space to leave open on each page (node) in the intermediate levels of the index. The PAD_INDEX option is useful only when FILLFACTOR is specified, because PAD_INDEX uses the percentage specified by FILLFACTOR.

Before using FILLFACTOR and PAD_INDEX, you should remember that reads tend to outnumber writes, even in an online transaction processing (OLTP) system. Using a low FILLFACTOR tends to increase the reads because it spreads tables over a larger number of pages, thereby reducing data density. Before using FILLFACTOR and PAD_INDEX, you should use Performance Monitor (Perfmon) to compare SQL Server reads to SQL Server writes, and then change the FILLFACTOR and PAD_INDEX settings only if writes are more than 30 percent of reads.

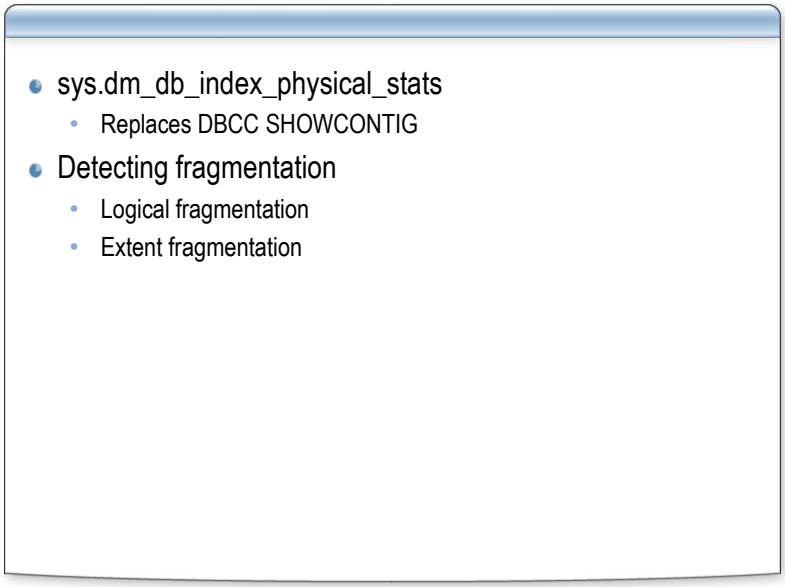
## Disabled Indexes

Disabling an index is beneficial in situations where you want to perform some administrative tasks that require you to first drop and then re-create the index. Now, you can simply disable, load, and rebuild the index instead. This makes the syntax and automation scripts a lot easier to write. Disabled indexes are not used or maintained. Disabling an index deallocates the storage space, but retains the metadata. Disabled indexes must be rebuilt before they can be used again.Disabling the clustered index makes the table unusable until the index is rebuilt.

To see if an index is disabled or not, you can use the sys.index catalog as follows:

```
select name, is_disabled from AdventureWorksPTO.sys.indexes
```

# Detecting Index Fragmentation




Indexes need routine maintenance to continue operating optimally. One of these routine maintenance tasks is Defragmenting. A fragmented index can have a negative impact on the efficiency of data access.

## dm_db_index_physical_stats

The **dm_db_index_physical_stats** dynamic management view (DMV) returns the size and fragmentation information for the data and indexes of the specified table or view. You should use this DMV to identify fragmentation. It replaces DBCC SHOWCONTIG from SQL Server 2000.

The **dm_db_index_physical_stats** DMV returns the following information:

- For an index, one row for each level of the B-tree in each partition.
- For a heap, one row for the IN_ROW_DATA allocation unit of each partition.
- For LOB data, one row for the LOB_DATA allocation unit of each partition.
- If row-overflow data exists in the table, one row for the ROW_OVERFLOW_DATA allocation unit in each partition.

By default, this DMV scans the page chain at the leaf level of the specified index. The following is the options to be used with calls to this function.

```
sys.dm_db_index_physical_stats (
{ database_id | NULL }, { object_id | NULL },
{ index_id | NULL | 0 }, { partition_number | NULL },
{ mode | NULL | DEFAULT } )
```

> **Note**: DBCC SHOWCONTIG will be removed in a future version of SQL Server. Avoid using this feature in new development work, and plan to modify applications that currently use this feature.

The following table describes the possible values for the mode option in the syntax of the dm_db_index_physical_stats DMV:

| Mode Value | Description |
|---|---|
| LIMITED (default) | LIMITED mode is the fastest and scans the smallest number of pages. It scans all pages for a heap, but only scans the parent-level pages, which means, the pages above the leaf-level, for an index. |
| SAMPLED | SAMPLED mode returns statistics based on a one percent sample of all the pages in the index or heap. If the index or heap has fewer than 10,000 pages, DETAILED mode is used instead of SAMPLED. |
| DETAILED | Detailed mode scans all pages and returns all statistics. |

From LIMITED to SAMPLED to DETAILED, the modes are progressively slower, because more work is performed in each. To quickly gauge the size or fragmentation level of a table or index, you should use the LIMITED mode. It is the fastest and will not return a row for each non-leaf level in the IN_ROW_DATA allocation unit of the index.

Regardless of the mode specified, sys.dm_db_index_physical_stats requires only an Intent-Shared (IS) table lock, unlike DBCC SHOWCONTIG, which required a shared (S) table lock.

## Detecting fragmentation

Two types of fragmentation can exist on indexes:

- **Logical fragmentation**

  Logical fragmentation is the percentage of an index that consists of out-of-order pages in the leaf pages. An out-of-order page is the one for which the next page indicated in an IAM is different from the page pointed to by the next page pointer in the leaf page.

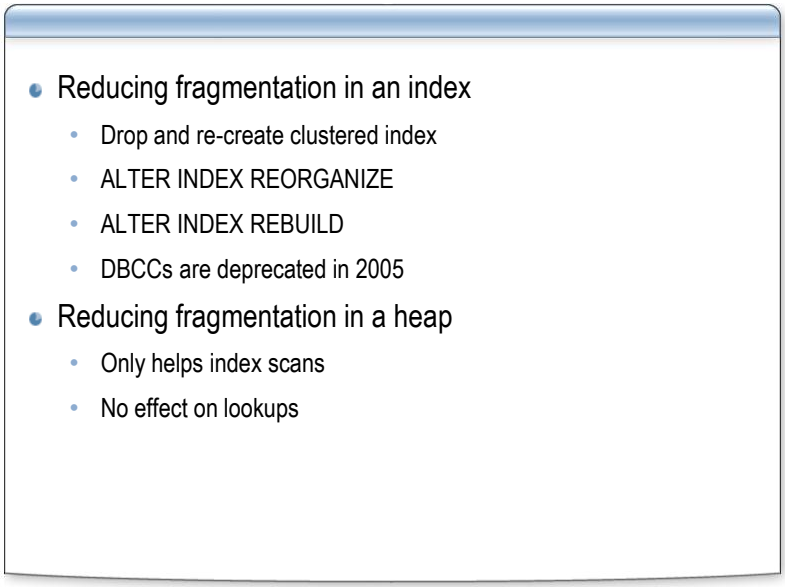
- **Extent fragmentation**

  Extent fragmentation is the percentage of a heap that consists of out-of-order extents in the leaf pages. An out-of-order extent is the one for which the extent that contains the current page for a heap is not, physically, the next extent after the extent that contains the previous page.

The fragmentation level of an index or heap is shown in the **avg_fragmentation_in_percent** column. For heaps, the value represents the extent fragmentation (equivalent to DBCC SHOWCONTIG Extent Scan Fragmentation). For indexes, the value represents the logical fragmentation (equivalent to DBCC SHOWCONTIG Logical Scan Fragmentation). Higher values for **avg_fragmentation_in_percent** indicate that the table or index is fragmented. For

maximum performance, the value for **avg_fragmentation_in_percent** should be as close to zero as possible.

# Reducing Fragmentation




## Reducing fragmentation in an index

When an index is fragmented in a way that affects the query performance, there are three choices for reducing fragmentation:

- Drop and re-create the clustered index.

  Re-creating a clustered index redistributes the data and results in full data pages. You can configure the level of fullness by using the FILLFACTOR option in CREATE INDEX. The drawbacks to this method are that the index is offline during the drop and re-create cycle, and the operation is atomic, which means that if the index creation is interrupted, the index is not re-created.

- Use ALTER INDEX REORGANIZE, the replacement for DBCC INDEXDEFRAG, to reorder the leaf-level pages of the index in a logical order.

  Because this is an online operation, the index is available while the statement is running. The operation can also be interrupted without losing work that has already been completed. The drawback to this method is that it does not do as good a job of reorganizing the data as the index rebuild operation, and it does not update statistics.

- Use ALTER INDEX REBUILD, the replacement for DBCC DBREINDEX, to rebuild the index online or offline.

  Fragmentation alone is not a sufficient reason to reorganize or rebuild an index. The main effect of fragmentation is that it slows down page read-ahead throughput during index scans. This causes slower response times. If the query workload on a

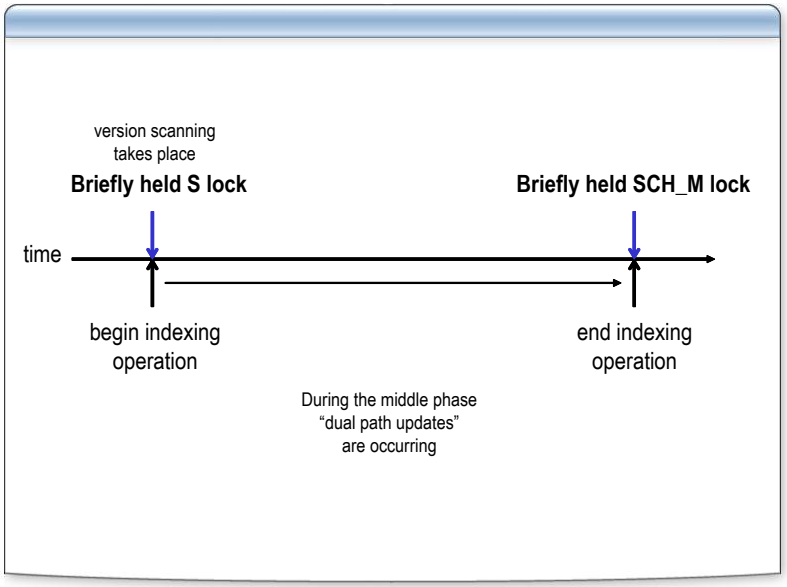
fragmented table or index does not involve scans, because the workload primarily consists of singleton lookups, removing fragmentation may not have any effect.

**Note**: DBCCs pertaining to indexes will be dropped in a future version of SQL Server. We recommend you to stop using them now.

## Reducing fragmentation in a heap

To reduce the extent fragmentation of a heap, you should first create a clustered index on the table and then drop the index. This redistributes the data while the clustered index is created. This also optimizes the data as much as possible, thereby considering the distribution of free space available in the database. When the clustered index is then dropped to re-create the heap, the data is not moved and remains in an optimal position.

# Online Index Maintenance




In the previous versions of SQL, index rebuilding operations required exclusive table locks for the duration of the indexing operation. Beginning with SQL Server 2005, indexes can be rebuilt online, which means that users can continue to access and modify data while the indexes are being rebuilt.

Online index operations allow for the following concurrent modification of the underlying table or index:

- Updates
- Inserts
- Deletes

You can perform online index maintenance by using the following commands:

- CREATE
- REBUILD
- REORGANIZE
- DROP

An online index operation:

1. Briefly acquires a shared lock at the table level to freeze out write plans momentarily.
2. Creates the metadata for the new index or heap by making it write-only, or not selectable as a data access path in a query plan.
3. Increments the schema version number.

4. Drops the shared lock and allows write plans back in.

All write plans are recompiled on first use.

If you are creating a non-clustered index (the simplest case), the base table is scanned by using a versioned scan (to get a snapshot of the data when the index DDL operation starts). In addition, all updates to the base table are reflected as normal in the in-build index.

If you are rebuilding a non-clustered index, the new in-build index is maintained, automatically, while the old index is using versioned scan to scan the base table. All updates to the base table are reflected as normal in the in-build index.

If you are rebuilding a clustered index, without changing the keys, the query plan uses a dual-update path in which data record changes are reflected both in the new in-build index and the old index at the same time.

You enable or disable table locks by using the ONLINE option as follows:

```
ONLINE = ON | OFF
```

- ON

  Long-term table locks are not held for the duration of the index operation. During the main phase of the index operation, only an IS lock is held on the source table. This enables queries or updates to the underlying table and indexes to proceed. At the start of the operation, an S lock is held on the source object for a very short time. At the end of the operation, for a short time, an S lock is acquired on the source if a non-clustered index is being created; or a Schema Modification (SCH-M) lock is acquired if a clustered index is being created or dropped online, and when a clustered or non-clustered index is being rebuilt. ONLINE cannot be set to ON when an index is being created on a local temporary table.

- OFF (default)

  The default position for table locks is OFF. Table locks are applied for the duration of the index operation. An offline index operation that creates, rebuilds, or drops a clustered index, or rebuilds or drops a non-clustered index, acquires an SCH-M lock on the table. This prevents all users from accessing the underlying table for the duration of the operation. An offline index operation that creates a non-clustered index acquires an S lock on the table. This prevents updates to the underlying table, but allows read operations, such as SELECT statements to be run.

An ONLINE operation requires more disk space, because an additional copy of the index exists during the operation. However, the extra availability provided by not locking the table or index during the entire build or rebuild operation may easily justify the extra disk cost. Additionally, an ONLINE operation might require additional resources for sessions running INSERT, UPDATE, or DELETE queries, because these queries must maintain both the old index and the index being built online. At the beginning of the ONLINE operation, any plans that run INSERT, UPDATE, or DELETE against the target table are

invalidated and will be recompiled on next use (with a recompile reason of Schema Changed). This recompile is done so that the plan can be generated to maintain both the old and new indexes. At the end of the ONLINE operation, the INSERT, UPDATE, or DELETE plans that reference the target table no longer need to maintain the in-build index, so they will be marked for recompile.

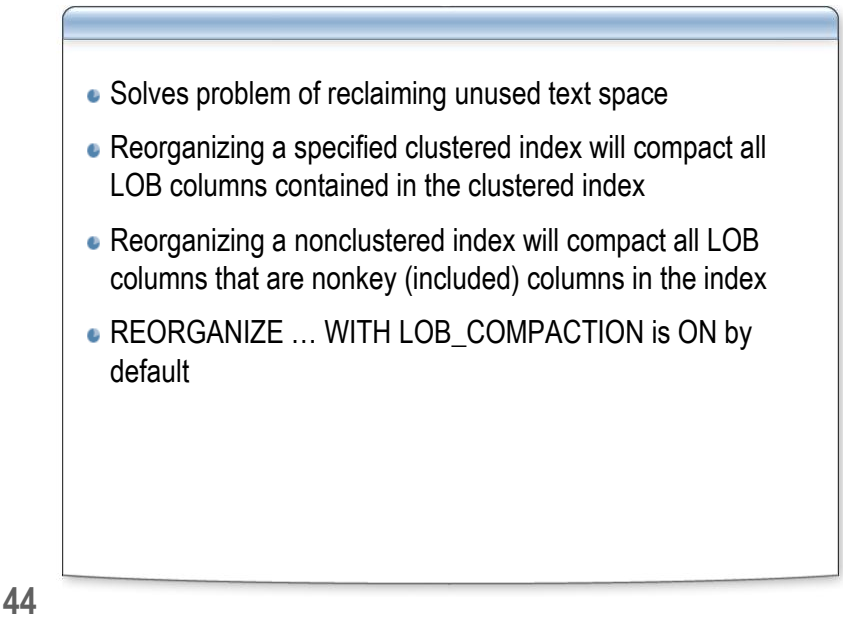
# Offline Only Index Operations

These cannot be done online:

- DBCC DBREINDEX
- Disabled clustered indexes – [create], [rebuild], or [drop]
- XML indexes – [create], [rebuild], or [drop]
- Indexed Views – [create]
- Nonclustered indexes – [drop]
- Nonclustered indexes with key or non-key columns of LOB data type – [rebuild]
  - image, ntext, text, varchar(max), nvarchar(max), varbinary(max), and xml
- Clustered indexes if the underlying table contains LOB data types – [rebuild]
  - image, ntext, text, varchar(max), nvarchar(max), varbinary(max), and xml

43

# LOB Compaction




When an index is reorganized, LOBs that are contained in the clustered index or underlying table are compacted, by default. The LOB data types include **image**, **text**, **ntext**, **varchar(max)**, **nvarchar(max)**, **varbinary(max)**, and **xml**. Compacting this data can result in better disk space use because:

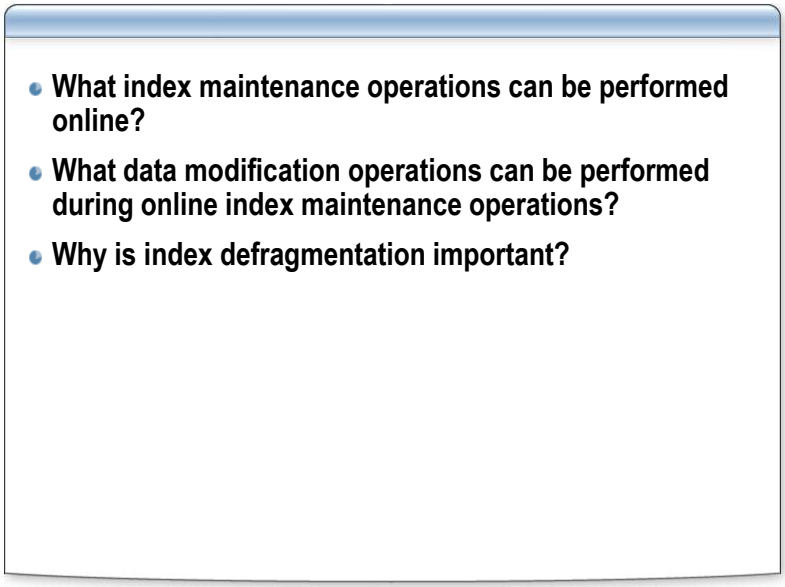
- Reorganizing a specified clustered index will compact all LOB columns that are contained in the leaf level (data rows) of the clustered index.
- Reorganizing a non-clustered index will compact all LOB columns that are non-key (included) columns in the index.
- When ALL is specified (as in "ALTER INDEX ALL ON <tablename> REORGANIZE"), all indexes associated with the specified table or view are reorganized, and all LOB columns associated with the clustered index, underlying table, or non-clustered index, with included columns are compacted.

When SQL Server compacts LOBs, it:

1. Locates all low-density (less than 75 percent used) extents.
2. Calculates an allocation threshold based on the number of low-density extents required to store all low-density text.
3. Scans the index by looking for text stored above threshold point, and moves it below the threshold.
4. Deletes the empty pages.

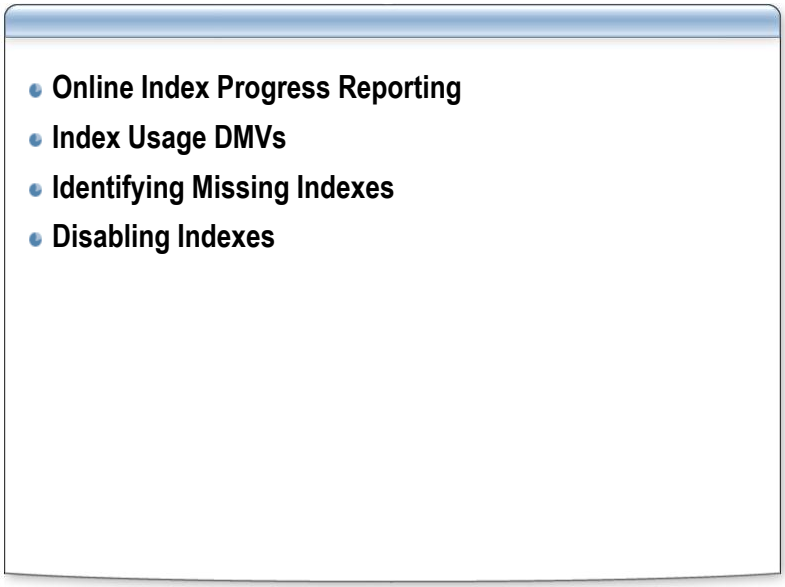
**Note**: The LOB_COMPACTION clause is ignored if LOB columns are not present.

## Section 5 Review




- What index maintenance operations can you perform online?
- What data modification operations can you perform during online index maintenance operations?
- Why is index defragmentation important?

# Section 6: Index Reporting and Monitoring




## Introduction

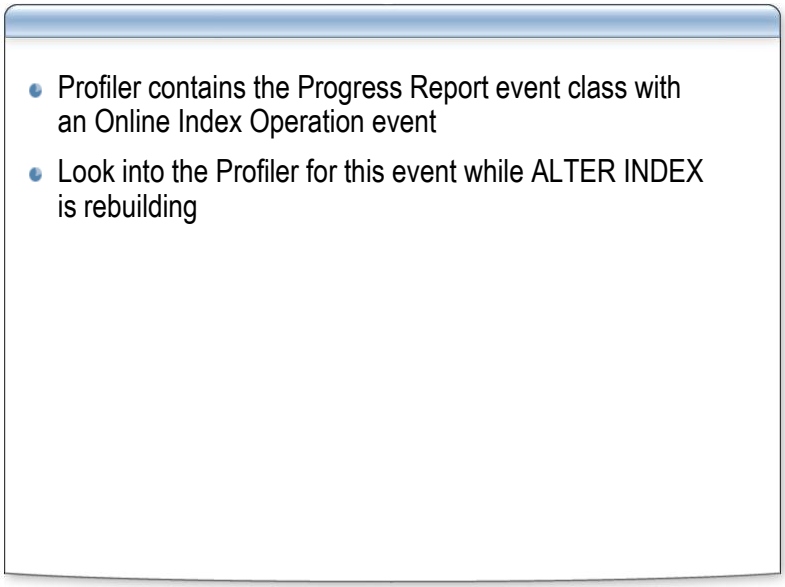
Beginning with SQL Server 2005, new tools were made available for monitoring index maintenance and usability. This section describes these index reporting and monitoring tools.

## Objectives

After completing this section, you will be able to:

- Locate the progress of online indexing operations.
- Track contention and I/O activity on an index.
- Determine the usefulness and usage history of an index.
- Identify missing indexes in a table.
- Disable an index.

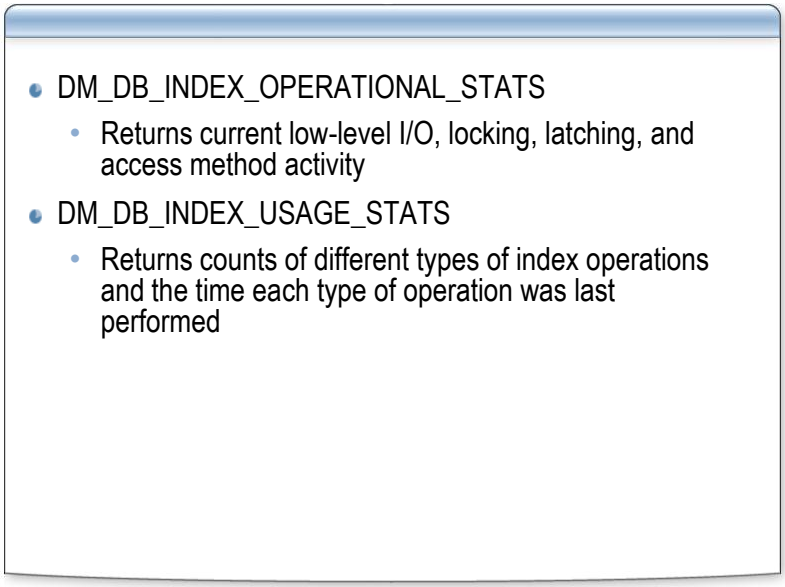
# Online Indexing Progress Reporting




The **Progress Report** event category in Profiler contains the **Progress Report: Online Index Operation** event class. This category indicates the progress of an online index build operation.

Indexing can produce large amounts of log records, and it also consumes CPU and disk resources as it progresses. As a result, there are times when evaluations must be made on other activity on the server, or even whether or not to kill the indexing process. The **Progress Report** event category can provide information to help make an informed decision in such times.

# Index Usage DMVs




In previous versions of SQL, it was not possible to see if indexes were being used, and if so, how useful they were. SQL Server 2005 introduced the following dynamic management functions (DMF) are available to give you insight into index usage:

### dm_db_index_operational_stats

You can use the dm_db_index_operational_stats DMF to determine track contention and I/O activity on an index.

To identify latching and locking contention, use the following columns:

- **page_latch_wait_count and page_latch_wait_in_ms**

  These columns indicate whether there is latch contention on the index or heap and the amount of the contention.

- **row_lock_count and page_lock_count**

  These columns indicate how many times the database engine tried to acquire row and page locks.

- **row_lock_wait_in_ms** and **page_lock_wait_in_ms**

  These columns indicate whether there is lock contention on the index or heap and the amount of the contention.

To analyze statistics of physical I/Os on an index or heap partition, use the following columns:

- **page_io_latch_wait_count** and **page_io_latch_wait_in_ms**

These columns indicate whether physical I/Os were issued to bring the index or heap pages into memory and report the number of I/Os that were issued.

### dm_db_index_usage_stats

Columns for **sys.dm_db_index_operational_stats** are maintained based on execution behavior, whereas columns for **dm_db_index_usage_stats** are incremented each time the index is referenced in the query plan. Therefore, you should use **dm_db_index_usage_stats** to determine the relative effectiveness of indexes.

> **Note:** Remember that DMVs exist only in memory. Therefore, no indexes will show usage in this DMV immediately after a restart of SQL. The information in the sys.dm_db_index_usage_stats DMV should be viewed for unused indexes only after a wide range of activity in the current server session.

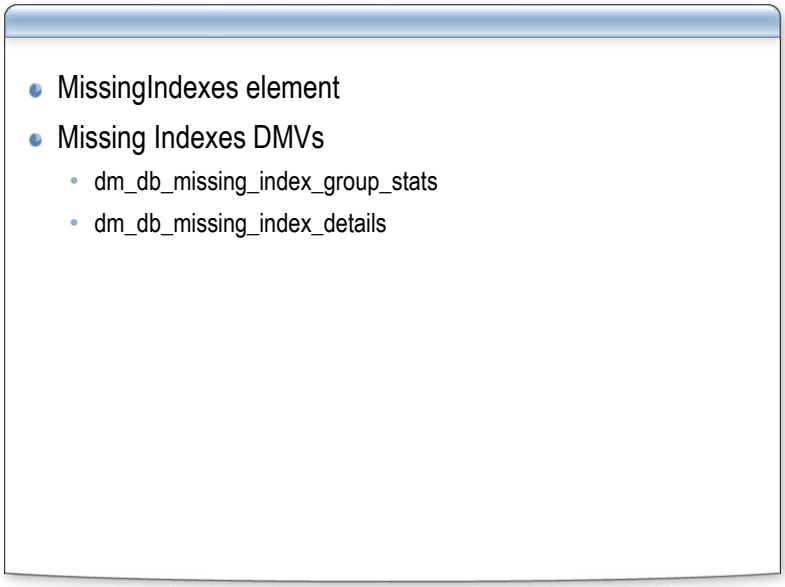
The following example shows how to find out unused indexes:

```
Use AdventureWorksPTO
Go

Declare @dbid int
Select @dbid = db_id('AdventureWorksPTO')

Select objectname=object_name(i.object_id), indexname=i.name, i.index_id
from sys.indexes i join sys.objects o on i.object_id = o.object_id
where objectproperty(o.object_id,'IsUserTable') = 1
and i.index_id NOT IN (select s.index_id
from sys.dm_db_index_usage_stats s
where s.object_id=i.object_id and i.index_id=s.index_id
and database_id = @dbid )
order by objectname,i.index_id,indexname asc
```

In the example shown below, rarely used indexes appear first:

```
Use AdventureWorksPTO
Go

declare @dbid int

select @dbid = db_id()

select objectname=object_name(s.object_id), s.object_id, indexname=i.name,
i.index_id
,       user_seeks, user_scans, user_lookups, user_updates
from sys.dm_db_index_usage_stats s join sys.indexes i
on i.object_id = s.object_id and i.index_id = s.index_id
where database_id = @dbid and objectproperty(s.object_id,'IsUserTable') = 1
order by (user_seeks + user_scans + user_lookups + user_updates) asc
```

# Identifying Missing Indexes




Indexes are crucial for efficient access of data from databases. Missing indexes can cause inefficient query plans, high number of scans producing high numbers of locks, high CPU usage, and memory pressure among other things. Beginning with SQL Server 2005, SQL tracks indexes that would have improved queries had those indexes existed at the time the queries were issued.

## MissingIndexes element

You can view the **MissingIndexes** element in the XML Showplan to determine if an index key column is used for equality (=) or inequality (<, >, and so on) in the Transact-SQL (T-SQL) statement predicate, or if it is just included to cover a query. The **MissingIndexes** element displays this information as one of the following values for the Usage attribute of the **ColumnGroup** sub-element:

- <ColumnGroup Usage="EQUALITY">
- <ColumnGroup Usage="INEQUALITY">
- <ColumnGroup Usage="INCLUDE">

## Missing indexes DMVs

The **MissingIndexes** element is useful for a specific query. The DMVs are cumulative for a workload. SQL Server resets the missing indexes DMVs when you restart it. Statistics are gathered for a maximum of 500 missing index groups. After this threshold is reached, no more missing index group data is gathered. However; if a new index would improve a query by more than 90 percent, and row that would only improve a query by a

small amount will be removed to make room for the new index group entry. This threshold is not a tunable parameter and cannot be changed.

The missing indexes feature is a lightweight tool for finding missing indexes that might significantly improve query performance. It does not provide adequate information to fine tune your indexing configuration. You should use the DTA for that purpose. You can turn off **gather_statistics_on_missing_indexes** only by running SQL Server from a command line by using the –x parameter.

> **Note:** the -x parameter also disables all SQL performance counters.

The following query determines which missing indexes would produce the highest anticipated cumulative improvement, in descending order, for user queries:

```
SELECT TOP 50 priority = avg_total_user_cost *
               avg_user_impact * (user_seeks + user_scans)
,      d.statement
,      d.equality_columns
,      d.inequality_columns
,      d.included_columns
,      s.avg_total_user_cost
,      s.avg_user_impact
,      s.user_seeks, s.user_scans
       FROM sys.dm_db_missing_index_group_stats s
              JOIN sys.dm_db_missing_index_groups g
                     ON s.group_handle = g.index_group_handle
              JOIN sys.dm_db_missing_index_details d
                     ON g.index_handle = d.index_handle
ORDER BY priority DESC
```

## Try This

Open the Query window in SQL Server Management Studio and try this:

```
USE AdventureWorksPTO;
GO


IF EXISTS (SELECT name FROM sys.indexes
           WHERE name = N'IX_PersonAddress_StateProvinceID')
     DROP INDEX IX_PersonAddress_StateProvinceID ON Person.Address;
GO

SELECT City, StateProvinceID, PostalCode
FROM Person.Address
WHERE StateProvinceID = 9;
GO

SELECT *
FROM sys.dm_db_missing_index_details
```

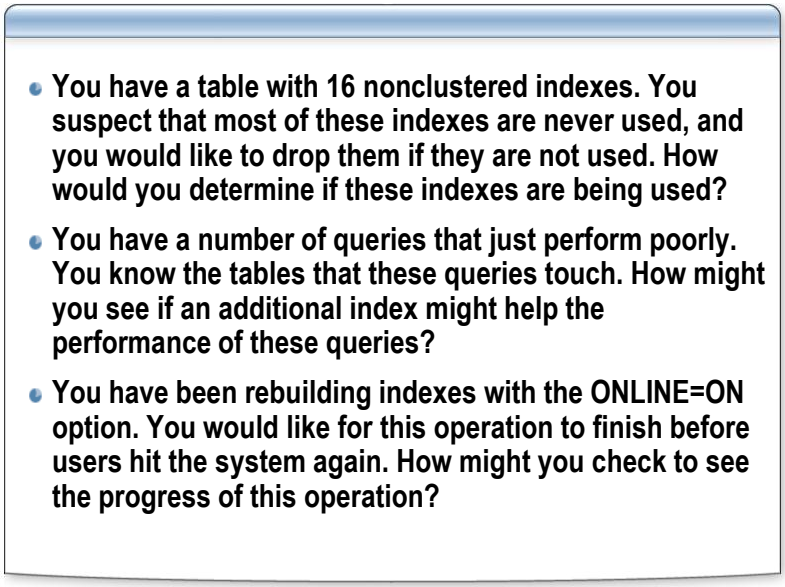
The example shown above should produce a row indicating a missing index on Person.Address.StateProvinceID. You can use that information to create the following index to cover the query:

```
USE AdventureWorksPTO;
GO
CREATE NONCLUSTERED INDEX IX_PersonAddress_StateProvinceID
     ON Person.Address (StateProvinceID)
     INCLUDE (City, PostalCode);
GO
```

Use the following guidelines to order columns in the CREATE INDEX statements that you write above:

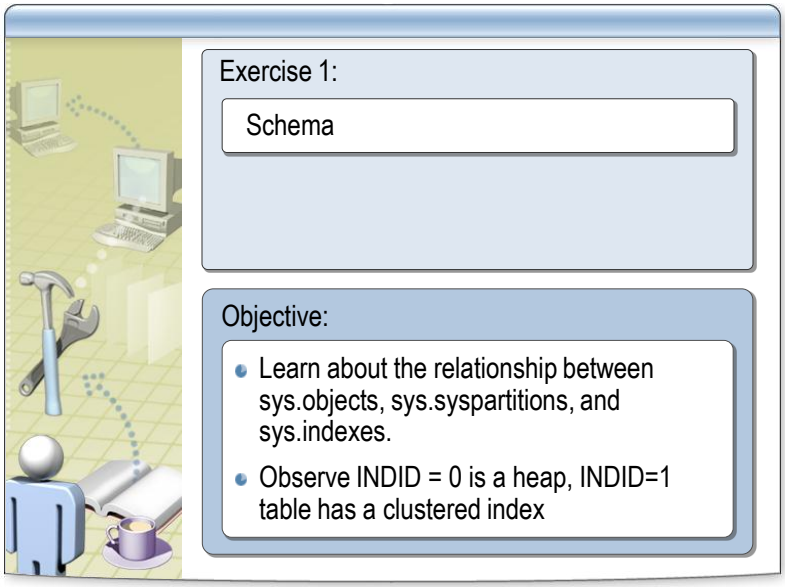
- List the equality columns leftmost in the column list.
- List the inequality columns to the right of equality columns.
- List the include columns in the INCLUDE clause of the CREATE INDEX statement.
- To determine an effective order for the equality columns, order them based on their selectivity, that is, list the most selective columns first.

# Section 6 Review

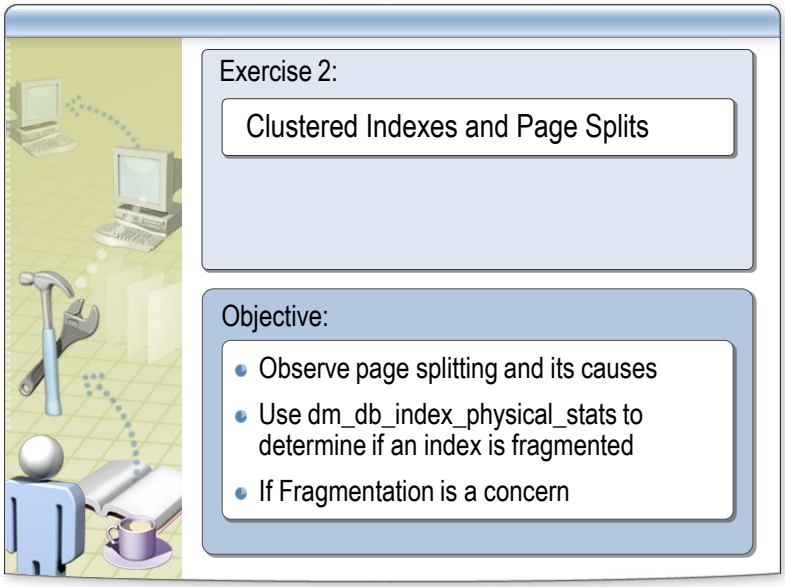



- You have a table with 16 non-clustered indexes. You suspect that most of these indexes are never used, and you would like to drop them. How would you determine if these indexes are being used?
- You have a large number of queries that perform poorly. You know the tables that are accessed by these queries. How can you determine if an additional index might help the performance of these queries?
- You are rebuilding indexes by using the ONLINE=ON option, and you want this operation to finish before users hit the system again. How can you check the progress of this operation?

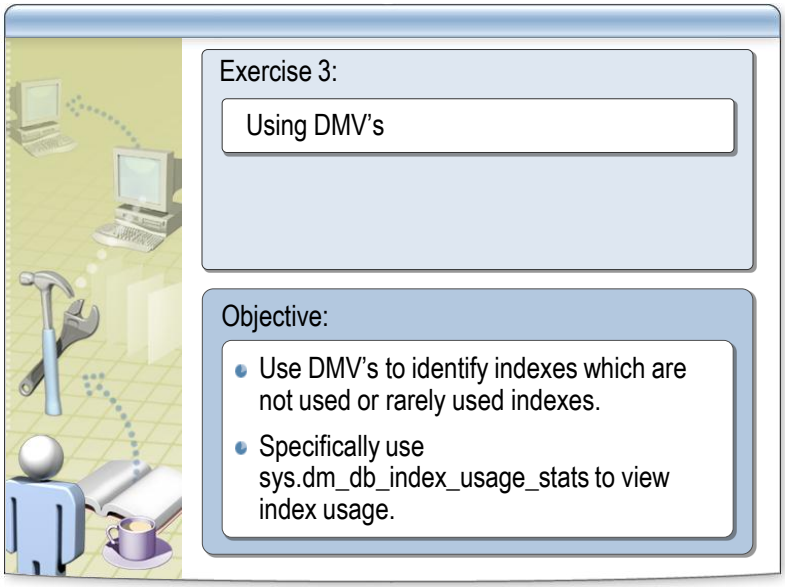
# Lab 1: Exercise 1



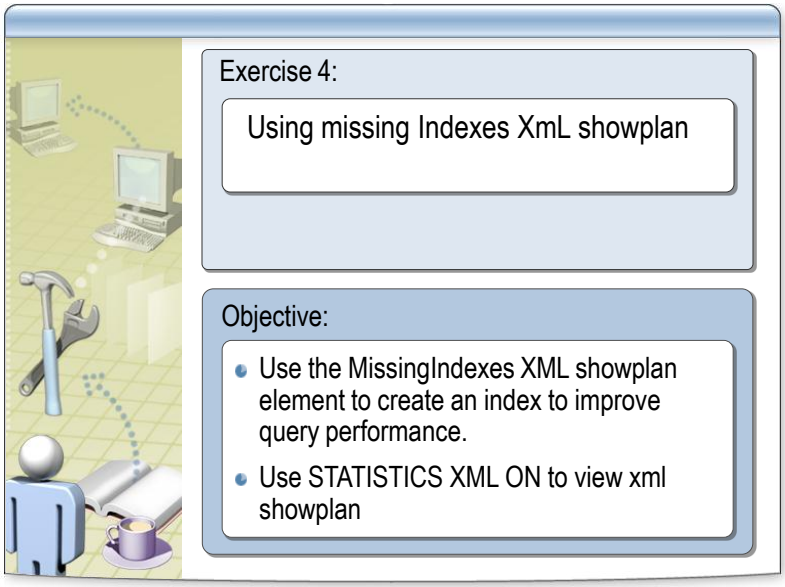
# Lab 1: Exercise 2



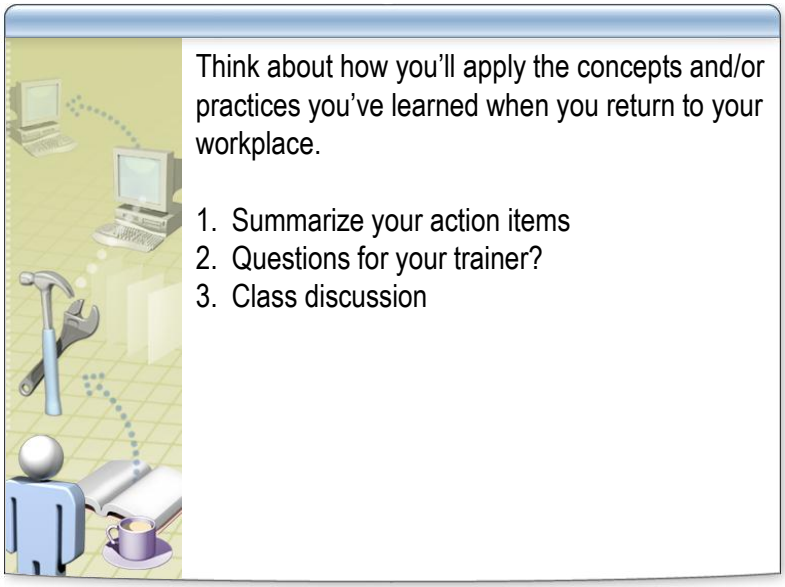
# Lab 1: Exercise 3



# Lab 1: Exercise 4



# Action Planning Exercise




When you return to your workplace, you'll want to use the concepts and practices you've learned to positively impact your IT environment. In this exercise you'll think about what you've learned and create action items that you can follow up on when you return to your workplace.

## Step 1: Summarize Your Action Items (5 Minutes)

How can you apply what you've learned to your workplace?

| | |
|---|---|
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |

## Step 2: Questions for your trainer? (5 Minutes)

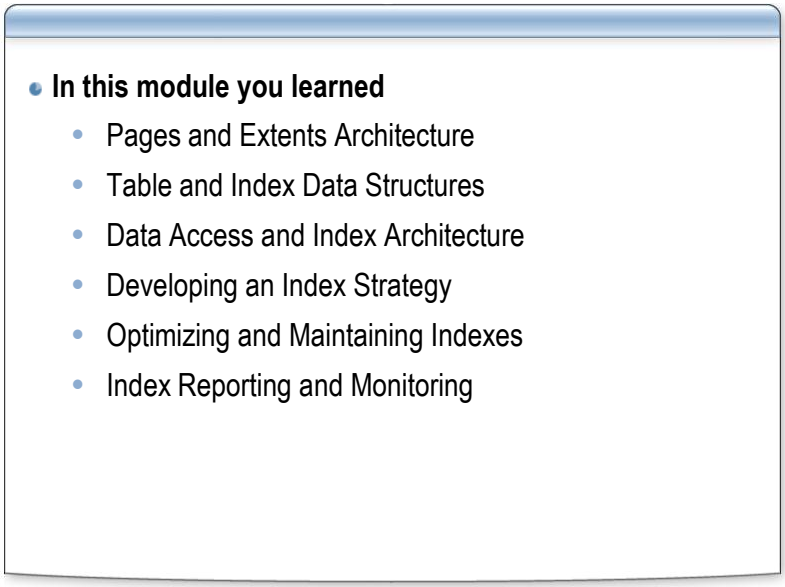
Record any questions you have about accomplishing the Action Items you listed above. The trainer will allow time for discussion before moving to the next module.

| | |
|---|---|
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |

## Step 3: Class Discussion

Notes:

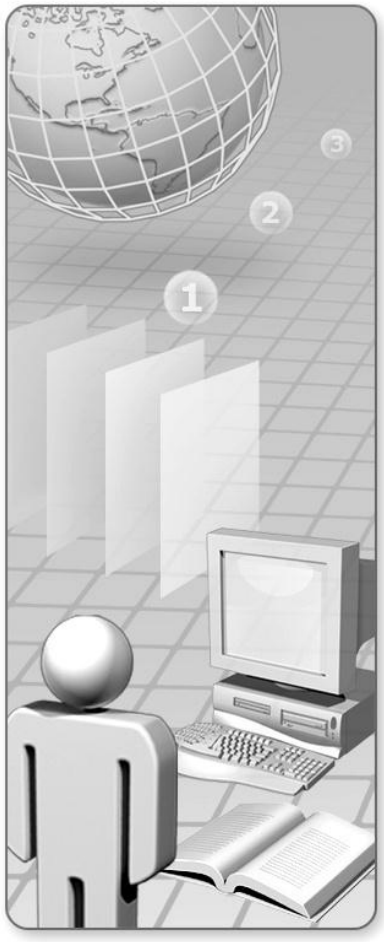
# Module Summary



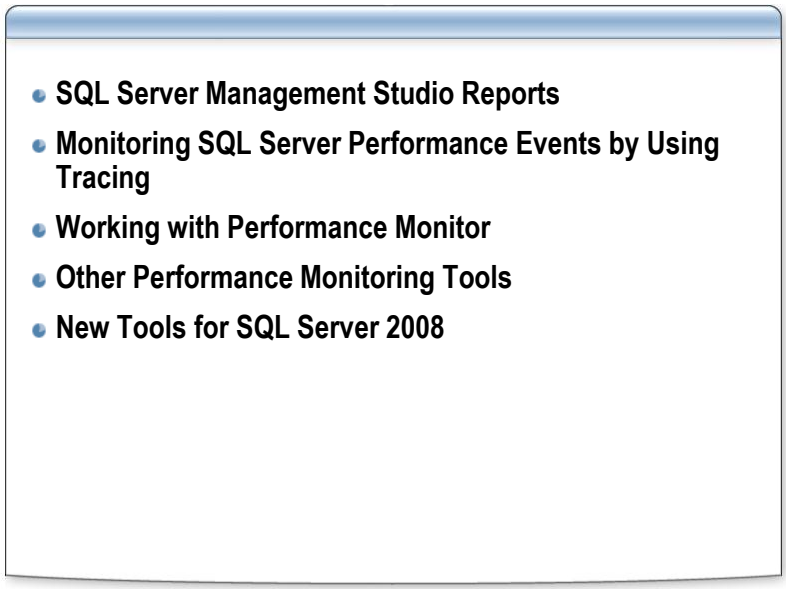

In this module, you learned about index physical and logical structures and how these structures are used for efficient data access. You also learned how to design an optimal index strategy for a given load, and how to maintain and evaluate the usefulness of your indexes.

Microsoft | Services

# Module 3: Performance Tools and Monitoring



# Module Overview




## Introduction

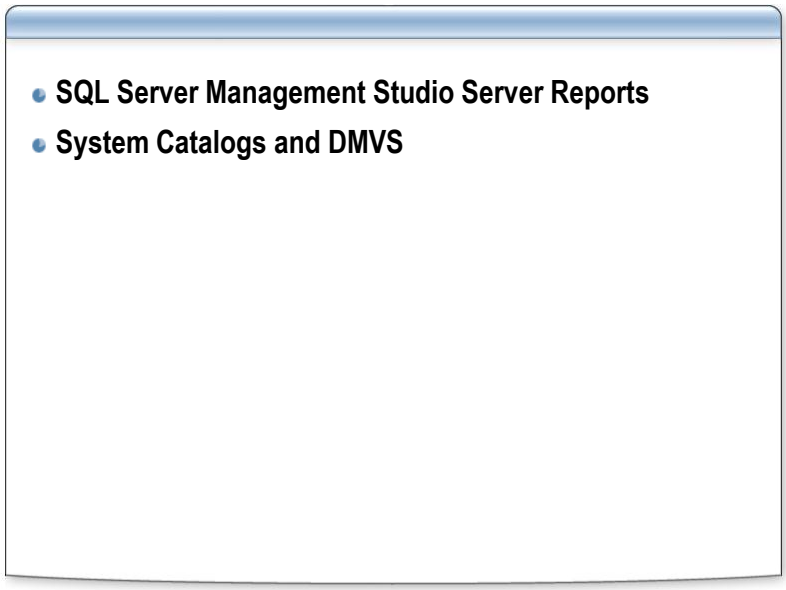
This module covers the tools that are used to troubleshoot performance issues in SQL Server. The goal of this module is to describe when to use each tool, how to configure each tool, and analyze the data captured.

## Objectives

After completing this module, you will be able to:

- Identify the tools used to troubleshoot performance issues in SQL Server and which tools can be used for real-time troubleshooting.
- Configure the performance analysis tools and analyze the data collected.
- Describe the performance reports available in SQL Server Management Studio.
- Monitor SQL Server performance events by using tracing and Performance Monitor (Perfmon).
- Collect performance data by using other tools, including the SQLDiag utility and the SQL Server Database Engine Tuning Advisor.

# Section 1: SQL Server Management Studio Reports




## Introduction

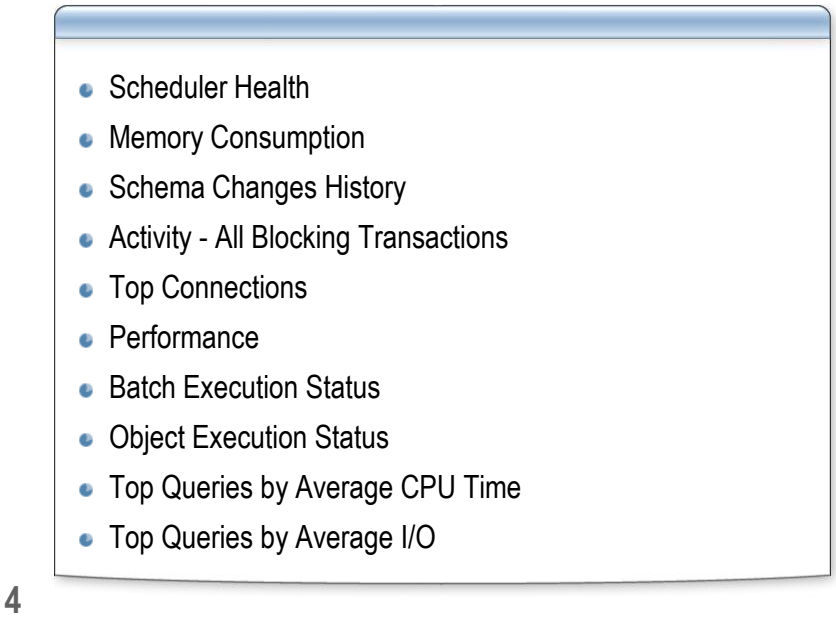
This section describes the built-in reports available in SQL Server Management Studio, and how these can be used for the real-time troubleshooting performance issues. The section also explains how to use dynamic management views (DMVs) for the real-time troubleshooting of performance issues.

## Objectives

After completing this section, you will be able to:

- Explain the purpose of the reports available in SQL Server Management Studio.
- Explain the role of DMVs in viewing SQL Server connections and identifying waits and queues.

# SQL Server Management Studio Server Reports



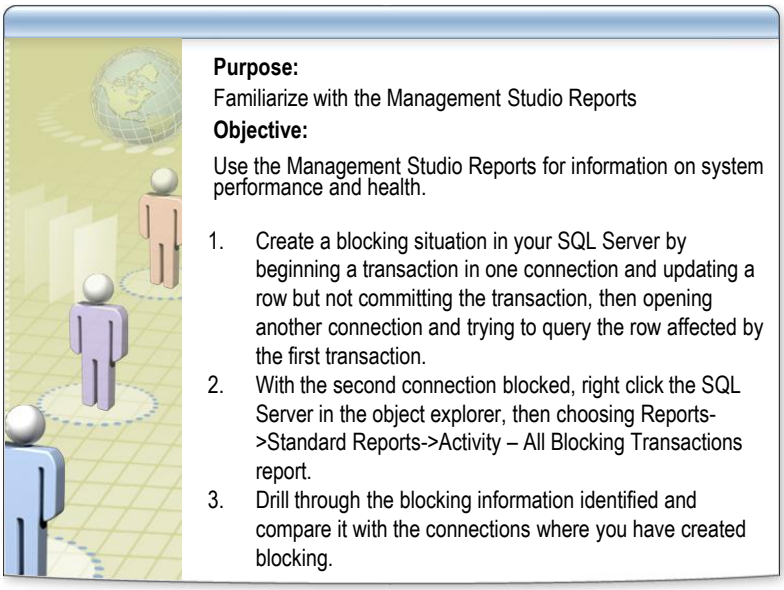

SQL Server Management Studio is an integrated environment for accessing, configuring, managing, administering, and developing all components of SQL Server. It replaces the Enterprise Manager and Query Analyzer available in the previous versions of SQL Server.

SQL Server Management Studio contains built-in performance reports at both server level and database level. Most reports are based on dynamic management views (DMVs). The following table describes the server-level reports available in SQL Server Management Studio:

| Report | Description |
|---|---|
| Scheduler Health | Provides information on each thread assigned to a scheduler. Replaces DBCC UMSSTATS in SQL Server 2000. |
| Memory Consumption | Provides information on the memory consumption of components within a SQL Server instance and the history of memory changes for the instance. |
| Schema Changes History | Provides a history of all the committed data definition language (DDL) statements. This information can be used to determine if indexes were created, dropped, and so on. |
| Activity - All Blocking Transactions | Reports any transactions that are currently blocked. |
| Top Connections | Provides a list of top ten connections based on age and input/output (I/O). |
| Performance Batch Execution Status | Provides execution data (CPU and I/O usage) for all currently cached batch plans. |
| Performance Object | Provides execution data (CPU and I/O usage, number of times |

| Report | Description |
|---|---|
| Execution Status | executed) for all currently cached object plans. |
| Top Queries by Average CPU Time | Reports queries in the current plan cache that have most heavily used the CPU on average each time they were executed. |
| Top Queries by Average I/O | Reports queries in the current plan cache that have used the most I/O on average each time they were executed. |

# Demonstration 1: Management Studio Reports



**Purpose:**

Familiarize with the Management Studio Reports

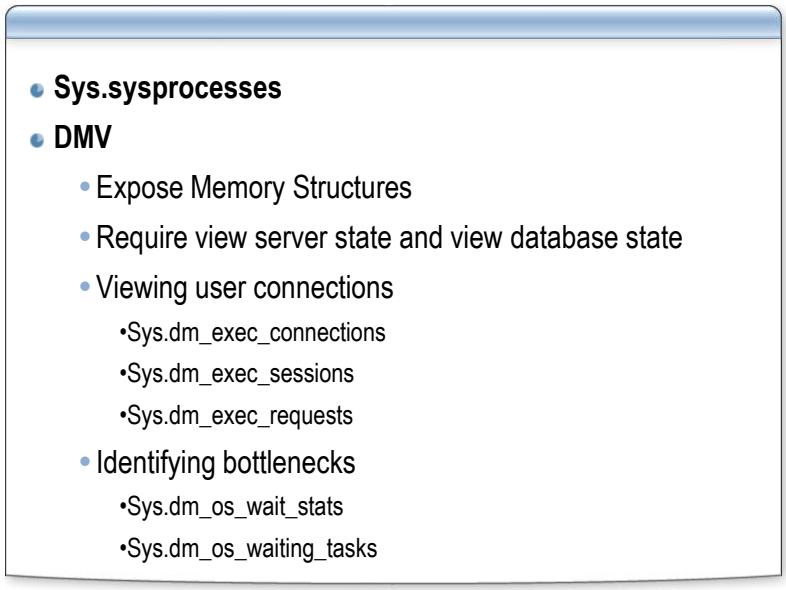
**Objective:**

Use the Management Studio Reports for information on system performance and health.

1. Create a blocking situation in your SQL Server by beginning a transaction in one connection and updating a row but not committing the transaction, then opening another connection and trying to query the row affected by the first transaction.
2. With the second connection blocked, right click the SQL Server in the object explorer, then choosing Reports->Standard Reports->Activity – All Blocking Transactions report.
3. Drill through the blocking information identified and compare it with the connections where you have created blocking.

5

# System Catalogs and DMVs




## Sys.sysprocesses

This view exists in the master database and returns one row for each process in SQL Server. You can use this view to troubleshoot performance issues. You should look for the following items in this view:

- Multiple rows with the same value for SPID can indicate the connection is probably running a parallel query.
- Rows with the blocked column non zero indicates that blocking is occurring. For example, if SPID 78 has 90 for the value in the blocked column, it implies that SPID 78 is being blocked by 90.
- Open transactions in the open_tran column indicates the nested transaction level. So, if open_tran is 2, the connection needs to issue two commit transaction statements before the locks are released. Also, if open_tran is greater than 0, you should check the status column to see if the connection is sleeping. If open_tran is greater than 0 and the status is ***sleeping***, it indicates that the SQL Server is waiting on the client to submit another query in the transaction, and then either commit or rollback the transaction.
- Status column indicates what the connection is doing. Possible values in the Status column include:
  - **dormant**: Indicates that the SQL Server is resetting the session.

- **running**: Indicates that the session is running one or more batches. When Multiple Active Result Sets (MARS) is enabled, a session can run multiple batches.

> **Note**: For more information, refer to SQL Server Books Online *Using Multiple Active Result Sets (MARS)*.

- **background**: Indicates that the session is running a background task, such as deadlock detection.
- **rollback**: Indicates that the session has a transaction rollback in process.
- **pending**: Indicates that the session is waiting for a worker thread to become available.
- **runnable**: Indicates that the task in the session is in the runnable queue of a scheduler while waiting to get a time quantum.
- **spinloop**: Indicates that the task in the session is waiting for a spinlock to become free.
- **suspended**: Indicates that the session is waiting for an event, such as I/O, to complete.

- Non-zero values for waittype indicate that the process is waiting on something. If you see a lot of SPIDs with the same waittype, it may indicate a bottleneck as the connections are waiting on the same resource.

> **Note**: For a list of waittypes, refer to ***sys.dm_os_wait_stats*** in SQL Server Books Online.

## DMVs

DMVs expose internal memory structures within the SQL Server address space. DMVs replace the virtual tables in SQL Server 2000 (ex sysprocesses). DMVs and dynamic management views functions (DMFs) reflect what is going on inside the server process itself or across all sessions in the server. DMVs can be used for diagnostics, memory and process tuning, and monitoring across all sessions in the server. DMVs alleviate the need to monitor the server with tools, such as Profiler and Performance Monitor (Perfmon), to diagnose performance issues. Additionally, DMVs can provide information about performance issues even after an issue has occurred.

### Permissions

DMVs are available at either the server level or the database level. For server-level DMVs, the information is collected since SQL Server was last started or since an object was cached in memory. For database-level DMVs (dm_db*), the information collected is since the last time SQL was started or the database was attached, brought online, or

restored. To enable users to access server-level DMVs, you must grant the ***view server state*** privilege. The example below shows how to set up this privilege:

```
use master
go
grant view server state to login
```

All DMVs and DMFs exist in the **sys** schema and follow the naming convention of **dm_category***. When you use a DMV or DMF, you must prefix the name of the view or function by using the **sys** schema. For example, the query to use the **dm_os_wait_stats** DMV is:

```
SELECT wait_type, wait_time_ms
FROM sys.dm_os_wait_stats
```

## Viewing user connections

### sys.dm_exec_sessions

This view contains one row for each session, uniquely identified by a session_id. A session is established when SQL Server validates the client's credentials and verifies that the client has access to the server. A **session_id** is equivalent to a **SPID** in prior versions (the SQL 2005 **@@SPID** global function returns the **session_id** of the caller).

There is a 1:1 relationship between sessions and connections.

> **Note**: In earlier versions of SQL Server, the system **SPIDs** (such as checkpoint, lazywriter, ghost record cleanup, etc.) always had a **SPID** value less than 50. In SQL Server 2005, however, this is no longer the case; a system **session_id (SPID)** and a user session can have any value.

You can distinguish between system and user sessions by using one of the following means:

- The value in the **sys.dm_exec_sessions.is_user_process** column.
- **Error messages**. Error messages have been changed so that the system session_ids have an "s" following the SPID. Below is an example error message from a system session:

```
2005-05-16 09:25:51.290  spid12s   Service Broker manager has started.
```

- The value in the **IsSystem** column available in trace events. A value of 1 in this column indicates a system SPID.

### sys.dm_exec_connections

This view contains one row for each connection with the SQL Server process. Each client will have at least one physical connection. Each row in this table is uniquely identified by a **connection_id** value, which is a unique identifier. This view does not return system-related connections.

If a client is using Multiple Active Result Sets (MARS), the client might have additional logical connections that are multiplexed on top of the physical connection. The **parent_connection_id** column is always *null* for the physical connection, while it is non-null for a logical connection.

There is a brief period where the connection has been made, but the client has not yet been authenticated. It is possible that the authentication may fail, and the client and/or server will close the connection. In this case, all rows in the **Sys.dm_exec_connections** view will have a corresponding session in **sys.dm_exec_sessions**. The **most_recent_sql_handle** column contains the last request executed on the connection. So, after a query has finished executing, you can use the query below to get the SQL statement last executed:

```
select text, * from sys.dm_exec_connections
cross apply sys.dm_exec_sql_text(most_recent_sql_handle)
```

### sys.dm_exec_requests

This view reports information such as the status (runnable, sleeping), the command type (SELECT, INSERT, etc.), the session that is blocking this request (if applicable), the waittype, etc., about currently active requests.

The vast majority of information that was available in **sysprocesses** in the older versions of SQL Server is now reported in **sys.dm_exec_requests**.

A request is considered to have a status of runnable when it is actually running (the current worker thread is running on the Server Operating System (SOS) scheduler) or eligible to run (for example, it has yielded and is waiting to run). It is also considered runnable if it has undergone a preemptive switch, although SOS has no idea whether the request is actually consuming CPU or is waiting on some external resource.

Normally, there is a 1:1 relationship between **sys.dm_exec_sessions** and **sys.dm_exec_requests**. However, if you are using MARS, or have a parallel query, then there can be a 1:many relationship between these two views.

> **Note**: The **sys.dm_exec_requests** view contains the **sql_handle** column, which is null unless the query is currently running. If a session enters a sleeping status due to a waiting command, no row is listed for the session in **sys.dm_exec_requests**.

Assuming session 54 is currently running, you can use the following query to obtain the SQL statement running:

```
declare @v1 varbinary(64)
select @v1= sql_handle from sys.dm_exec_requests where session_id=54
select * from sys.dm_exec_sql_text(@v1)
```

## Identifying bottlenecks

### sys.dm_os_wait_stats

This view has replaced DBCC SQLPERF(WAITSTATS) in SQL Server 2000. The view returns information about the waits encountered by threads that are in execution.

Specific types of wait times during query execution can indicate bottlenecks or stall points within the query. Similarly, high wait times, or server-wide wait counts can indicate bottlenecks or hotspots in query interactions within the server instance. For example, *lock waits* indicate data contention by queries, *page I/O latch waits* indicate slow I/O response times, and *page latch update waits* indicate incorrect file layout. The statistics returned indicate the wait information since SQL Server was last restarted. Therefore, you want to take snapshots of this DMV at different times and compare the differences.

For example, you query the DMV at 10 AM and waittype x shows 1000. At 10:15 AM, the value for waittype x is 1400. So, 1400-1000 is the time spent waiting on the resource in the last 15 minutes.

> **Note**: For a complete list of waittypes and their descriptions, refer to Books Online at http://support.microsoft.com/kb/822101/en-us.

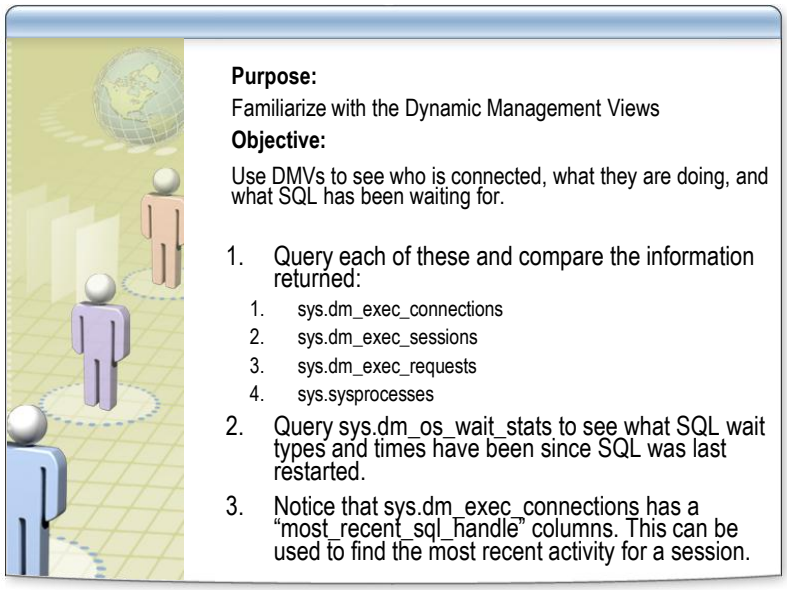
The **signal_wait_time_ms** column specifies the duration for which a task is in the runnable queue, waiting for CPU time. Long times spent in the runnable queue may indicate a CPU bottleneck. You can use the following query to determine the time spent in the runnable queue, waiting for other resources:

```
       Select signal_wait_time_ms=sum(signal_wait_time_ms),
      '%signal waits' = cast(100.0 * sum(signal_wait_time_ms) / sum (wait_time_ms)
as numeric(20,2))
               ,resource_wait_time_ms=sum(wait_time_ms - signal_wait_time_ms)
               ,'%resource waits'= cast(100.0 * sum(wait_time_ms -
signal_wait_time_ms) / sum (wait_time_ms) as numeric(20,2))
       From sys.dm_os_wait_stats
```

### sys.dm_os_waiting_tasks

This view returns sessions that are waiting on a resource. This view is similar to querying *sys.sysprocesses* and returning rows with either a non-zero **waittype** column or the **blocked** column greater than 0. In addition, the **sys.dm_os_waiting_tasks** view has been extended to show wait information for any arbitrary SQL resource, such as a latch, trace buffer, backup I/O, etc. Where possible, these waits are tied to the corresponding session (**sys.dm_exec_sessions**) via the **session_id** column. In the event of blocking, the **blocking_session_id** column will be greater than 0, and the **resource_description** column will contain additional information about the wait resource.

# Demonstration 2: Dynamic Management Views

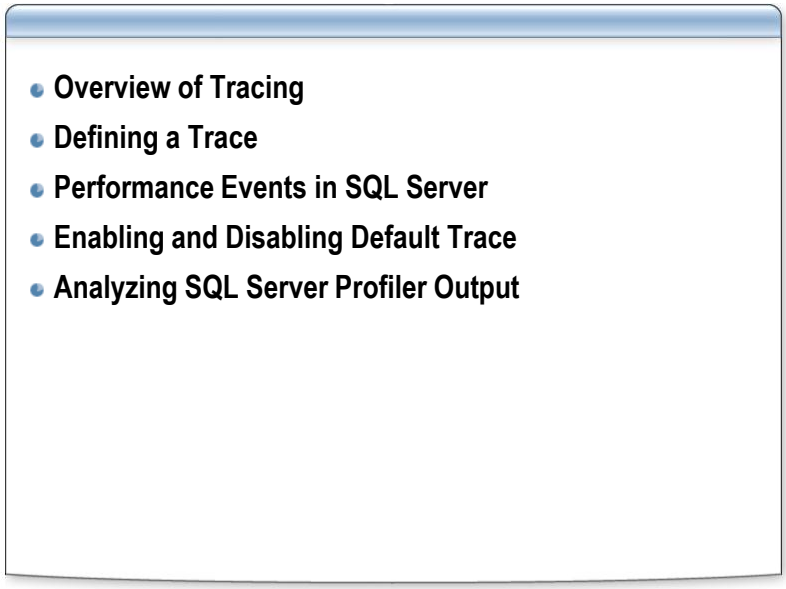


## Section 1 Review

- Which reports can be used to view CPU consumption, blocking, and expensive queries?
- Which report can be used to determine if an index has recently been dropped?
- What should you look for when running queries by using sys.sysprocesses?
- What should you look for when running queries against sys.dm_os_waiting_tasks?

8

# Section 2: Monitoring SQL Server Performance Events by Using Tracing




## Introduction

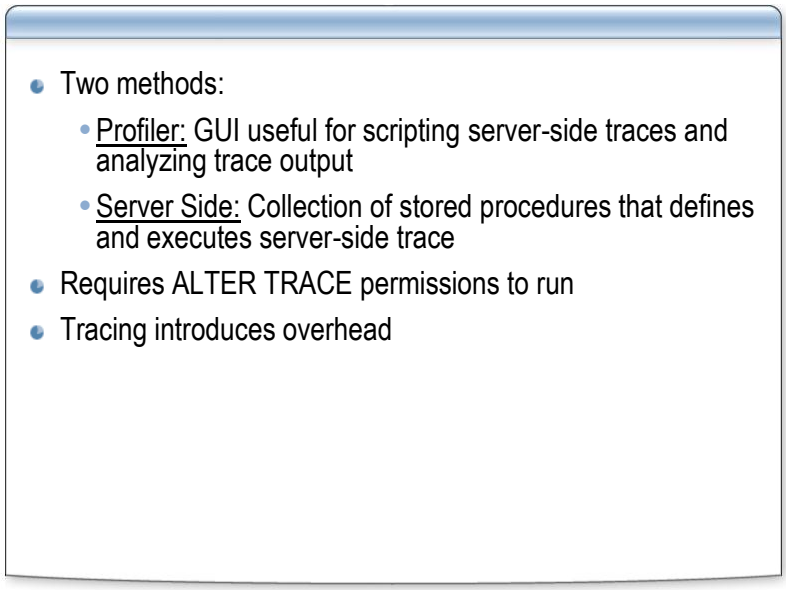
This section describes how to configure SQL Server Profiler to capture performance events. The section also explains how to analyze data and how to leverage the default trace.

## Objectives

After completing this section, you will be able to:

- Explain how SQL Server Profiler traces performance events in SQL Server.
- Define a trace in SQL Server Profiler.
- Identify the performance-related events in SQL Server.
- Enable and disable the default trace in SQL Server Profiler.
- Analyze SQL Server Profiler trace tables by using filtering and grouping.

# Overview of Tracing

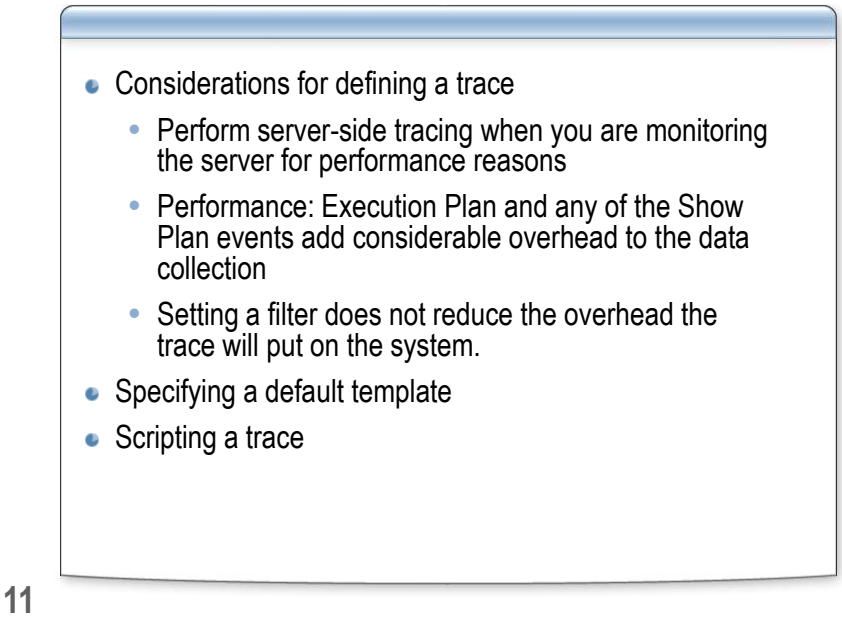



SQL Server Profiler is a graphical tool that enables you to monitor and collect server events. This information can be displayed in SQL Server Profiler, or saved as trace files, XML files, or in SQL Server tables.

The SQL Server engine can produce events in specific areas of interest. The SQL Server Profiler traces these events on the server side and then produces the data for a consumer, such as the SQL Server Profiler gui or a server-generated trace file. You can later analyze this data or replay a specific series of steps when trying to diagnose a problem.

SQL Server Profiler no longer requires the *system administrator* role in SQL Server. If a user is granted the ALTER TRACE permission, the user can start a SQL Profiler trace.

# Defining a Trace




To define a trace in SQL Server Profiler:

1. On the **File** menu, click **New Trace**, and connect to a SQL Server instance. The **Trace Properties** dialog box appears.
2. In the **Trace name** box, type a name for the trace.
3. In the **Use the template** list, select a trace template on which you want the trace to be based, or select **Blank** if you do not want to use a template.
4. Click **Save to file** to capture the trace to a file.
5. Specify a value of 500 megabytes (MB) or smaller for **Set maximum file size**.
6. Select **Enable file rollover** to automatically create new files when the maximum file size is reached.
7. You can also optionally select **Server processes trace data**, which causes events to be processed on the server instead of the client application.

   When the server processes trace data, no events are skipped even under stress conditions, but server performance may be affected.

   When this option is not selected, the processing is performed by the client application, and there is a possibility that some events will not be traced under stress conditions. Additionally, server-side tracing produces less overhead than performing graphical user interface (GUI) tracing. Therefore, if you are monitoring the server for performance reasons, we recommend performing server-side tracing.

8. To add or remove events, data columns or filters, click the **Events Selection** tab, and then perform the following steps:
   - To view a complete list of events, select the **Show all events** check box.
   - To view a complete list of columns, click **Show all columns**.

> **Warning:** Some events should be used cautiously, specifically, Performance: Execution Plan and any of the Show Plan events. These events add considerable overhead to the data collection, not only in the size of the event data produced and its impact on the trace file size, but also due to the high CPU overhead to generate the event.
>
> You may need to include these events when trying to get detailed performance traces, but avoid these events when the goal is to get an initial set of data to analyze. The more events you add, the more overhead and stress is put on SQL Server.
>
> The most important part of collecting Profiler information is determining which events and data columns to capture. The type of problem dictates which events are important to capture. Capturing Profiler data is often an iterative process, and the events captured may change as source of the problem is narrowed down. You may need to capture trace files multiple times to get the data needed to resolve a performance issue.

   - To specify a filter, click **Column Filters**.

     Character data columns allow you to specify LIKE and NOT LIKE filters. Integer columns allow you to specify EQUALS, NOT EQUAL TO, GREATER THAN OR EQUAL, or LESS THAN OR EQUAL filters.

     Setting a filter does not reduce the overhead that the trace will put on the system. The SQL Server Profiler captures all events and then applies the filter to determine if the event should be captured to the output.

## Specifying a default template

The default template is automatically selected when a new trace is created. To change the default template:

1. On the **File** menu, click **Templates**, and then click **Edit Template**.
2. In the **Trace Template Properties** dialog box, on the **General** tab, select a server type from the **Select server type** list.
3. In the **Select template name** list, select the name of the template that you want to use as the trace definition default.
4. Click **Use as a default template for selected server type**.
5. If necessary, click the **Events Selection** tab to modify the template.
6. Click **Save**.

## Scripting a trace

The recommend way to perform a server side trace is to script a trace in SQL Server Profiler:

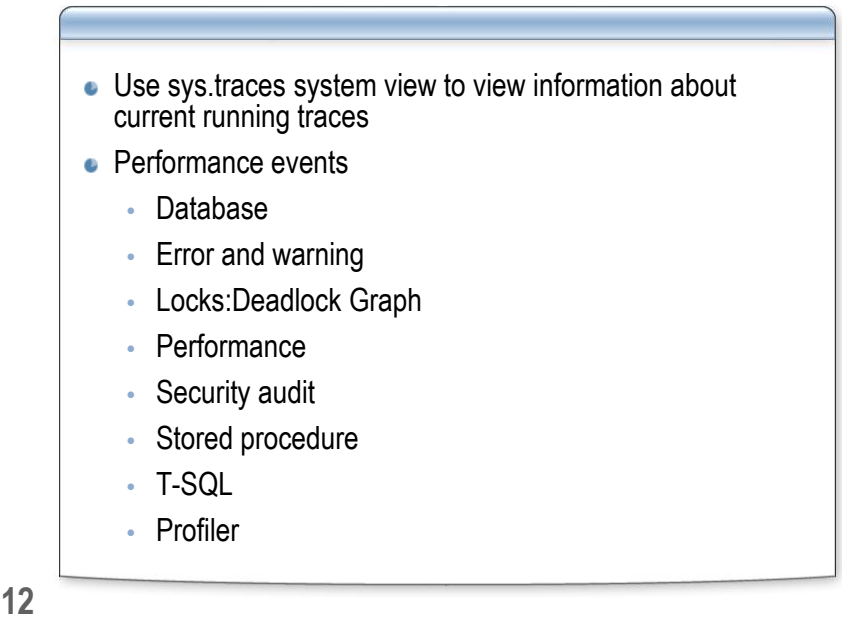
1. Define a trace.
2. Run the trace to verify that it contains the events that you need.
3. On the **File** menu, click **Export**, and then click **Script Trace Definition**. This creates a .sql file that contains the script.

You can then run the script by using SQL Server Management Studio. To start and stop the trace, use the stored procedure **sp_trace_setstatus**.

The following table describes the views that you can use to return information about traces, including the currently running traces and the full set of events that you can trace:

| View | Description |
|---|---|
| sys.traces | The **sys.traces** view returns information about the traces that are running; the current trace file name; the number and size of the trace buffers; options, such as stop time, rollover, max file size, default trace; etc. This view replaces fn_trace_getinfo available in earlier versions of SQL Server. |
| sys.trace_events | The **sys.trace_events** view returns the entire set of SQL Server events that can be traced, along with the name and category (used for grouping in Profiler) of each event. This view is useful to get the name of an event when you have the event ID. |
| sys.trace_categories | The **sys.trace_categories** view returns the category names (for example, T-SQL, Errors, Warnings, Stored Procedures, etc.) displayed in the SQL Server Profiler user interface for the different types of events. |

# Profiler Performance-Related Events




## Database events

SQL Server database events include:

- **Data File Auto Grow**
- **Data File Auto Shrink**
- **Log File Auto Grow**
- **Log File Auto Shrink**

These events occur when database file size changes automatically. While files are shrinking or growing, active transactions will wait for the event to finish. Therefore, changing file size can cause blocking and query timeouts, and negatively affect the performance.

## Error and warning events

The following table describes the error events:

| Event | Description |
|---|---|
| Attention | Indicates if there are queries being cancelled by users or applications. Excessive attention events might be an indication of poor query performance—users tired of waiting for their queries to return results. Attention events might also be an indication of blocking and query timeouts being reached. |
| Blocked Process Report | Occurs when a process has been blocked by a specified amount of time. The time is specified by using **sp_configure** to set the blocking process threshold. This event is recorded when blocking has exceeded the blocking process threshold. The TextData column of this event contains the output as |

| Event | Description |
|---|---|
| | XML. SQL Profiler and the Blocked Process Report event replace the Blocker script in SQL Server 2000. |
| ErrorLog | Is an event written to the SQL Server error log. |
| EventLog | Is an event written to the Windows Application event log. |
| Exception | Indicates that SQL Server returned an error to the client. |
| Exchange Spill Event | Indicates that communication buffers in a parallel query plan have been temporarily written to the **tempdb** database. This event occurs rarely and only when a query plan has multiple range scans. This event can indicate a slow-running query. |

You can use the following warning events to identify potential slow queries that are candidates to be optimized:

- **Execution Warnings**
- **Hash Warning**
- **Missing Column Statistics**
- **Missing Join Predicate**
- **Sort Warnings**

## Locks:Deadlock Graph event

The **Locks:Deadlock Graph** event captures deadlock information. The **TextData** column contains XML information that you can extract to a separate file. SQL Server Profiler shows the deadlock graphically; the node with an X indicates the deadlock victim. You should use this event instead of **Lock:Deadlock** and **Lock:Deadlock Chain** because the latter two events show details about the processes involved in deadlocks as hex information.

## Performance events

The following table describes the performance events:

| Event | Description |
|---|---|
| Auto Stats | Occurs with the automatic creation and updating of statistics. When statistics change for a table, query plans referencing the table are invalidated. Therefore, query plans need to be recompiled, which can drive up the CPU usage. |
| Showplan Statistics Profile | Displays the text-based query plan. This event can increase the size of the trace and overhead on the server. Therefore, you should use this event with caution and usually if you are only monitoring the server for a short time frame. |
| Showplan XML Statistics Profile | Graphically displays the execution plan. This event is similar to the Display Estimated Execution Plan in SQL Server 2000. The Showplan XML Statistics Profile event captures the same information as Showplan Statistics Profile, so you should use one or the other. The Showplan XML Statistics Profile event can increase the size of the trace and overhead on the server. Therefore, you must use this event with caution and usually if you are only monitoring the server for a short time frame. |
| Showplan All For | Is generated only when a cached plan is compiled. This event is different than |

| Event | Description |
|---|---|
| Query Compile | the Showplan All For Compile event, which is produced for every invocation of a query, regardless of whether it is using a cached plan or has compiled a new one. So, if the Showplan All For Query Compile event is being captured, but the event does not occur for a stored procedure call, it implies that the plan is cached. |
| Showplan XML for Query Compile | Is similar to Showplan All For Compile, but produces the query plan in a graphical format. |
| Performance Statistics | Indicates that a compiled plan has been cached for the first time, recompiled, or evicted from the plan cache. You can use the following **EventSubClass** values to determine if the query is in cache, is ad hoc, or is within a stored procedure:<br>• 0: Indicates that new SQL text is not currently present in the cache.<br>• 1: Indicates that queries within a stored procedure have been compiled.<br>• 2: Indicates that queries within an ad hoc SQL statement have been compiled.<br>• 3: Indicates that a cached query has been destroyed and the historical performance data associated with the plan is about to be destroyed.<br>The following EventSubClass types are generated in the trace for ad hoc batches and stored procedures:<br>• For ad hoc batches with n number of queries:<br>  • 1 of type 0<br>  • n number of type 2<br>  • 1 of type 3 when the query is flushed from the cache<br>• For stored procedures with n number of queries:<br>  • 1 of type 0<br>  • n number of type 1<br>  • 1 of type 3 when the query is flushed from the cache |

## Security audit events

The following security audit events indicate when a connection is made and closed:

- **Audit Login**
- **Audit Logout**

## Server:Server Memory Change event

This event indicates that the memory used by SQL Server has changed. Frequent occurring of this event indicates memory pressure on the server.

## Sessions:Existing Connection event

This event shows the properties of existing user connections when the trace was started. You can use this event to view the transaction isolation level being used by a connection.

## Stored procedure events

These events indicate when a stored procedure runs through a remote procedure call (RPC):

- **RPC:Completed**

- **RPC:Starting**

Stored procedure events show the individual SQL statements running inside the following stored procedures:

- **SP:StmtStarting**
- **SP:StmtCompleted**

> **Note:** Stored procedure events can increase the size of the trace and overhead on the server. Therefore, you must use them with caution.

The following stored procedure events indicate whether the execution plan for the stored procedure was found in cache, removed from cache, or inserted into cache. If the same stored procedure is called frequently, it should remain in cache. If the stored procedure is executed frequently and is not found in cache, that may indicate memory pressure either on the server or within SQL Server.

- **SP:CacheHit**
- **SP:CacheInsert**
- **SP:CacheMiss**
- **SP:CacheRemove**

### T-SQL events

SQL Server Profiler has the following T-SQL events:

- **SQL:BatchCompleted**
- **SQL:BatchStarting**

The above events indicate when a SQL batch begins and ends. A batch can be a single SQL statement or a group of statements.

### SQL:StmtRecompile

This event occurs when a statement inside a batch (including stored procedures) recompiles. In SQL Server 2005, if a statement within a stored procedure causes a recompile, only that statement is recompiled. In SQL Server 2000, if a statement within a stored procedure causes a recompile, the entire stored procedure is recompiled.

> **Note**: You should use the SQL:StmtRecompile event instead of SP:Recompile.

### Transaction:SQL Transaction event

This event tracks Begin, Commit, save, and Rollback transaction statements.

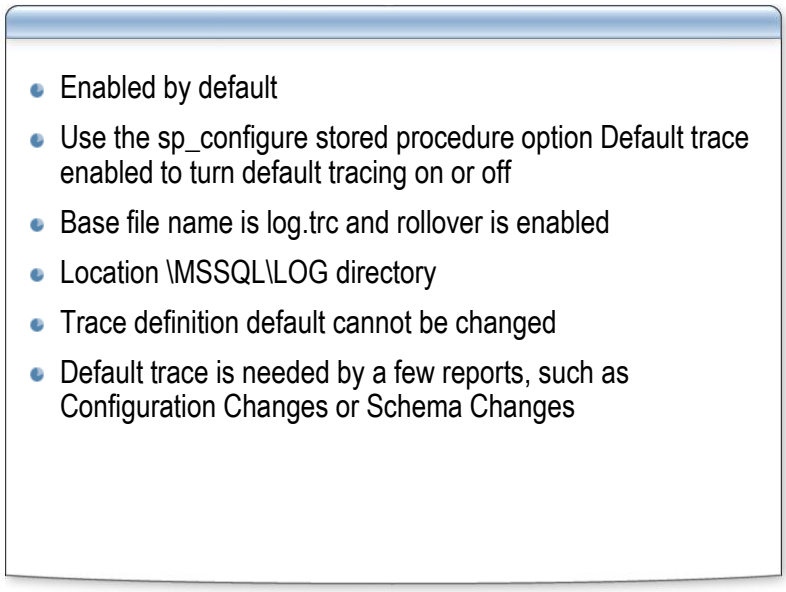
### Events Extraction Settings tab

If you choose any of the Performance:Showplan XML events or the Lock:Deadlock graph event, you will see an *Events Extraction Settings* tab. When you click on this tab,

you can choose to save those events automatically to either a single file or into multiple files by using the following options:

| Option | Description |
|---|---|
| Extract Show Plan Events | Provides the ability to save graphical query plans into .sqlplan files, which can be viewed later by using SQL Server Management Studio. This option requires the trace to include the Showplan XML Statistics Profile event. |
| Extract Deadlock Events | Provides the ability to save deadlock events to a separate deadlock XML (.xdl) file, which can be viewed by using SQL Server Management Studio. This option requires the trace to include the Deadlock Graph event. |

# Default Trace




Default tracing is enabled in SQL Server 2005 and above at all times.

Default trace captures schema and configuration changes. SQL Server maintains five default trace files and they are rolled over when the file size reaches 20 MB.

The following example uses the catalog views and functions to list the event IDs and event names that are part of the default trace:

```
select distinct e.trace_event_id, e.name
from sys.traces t
       cross apply ::fn_trace_geteventinfo(t.id) as ei
       join sys.trace_events e on ei.eventid = e.trace_event_id
where t.is_default = 0x1
order by e.trace_event_id
```

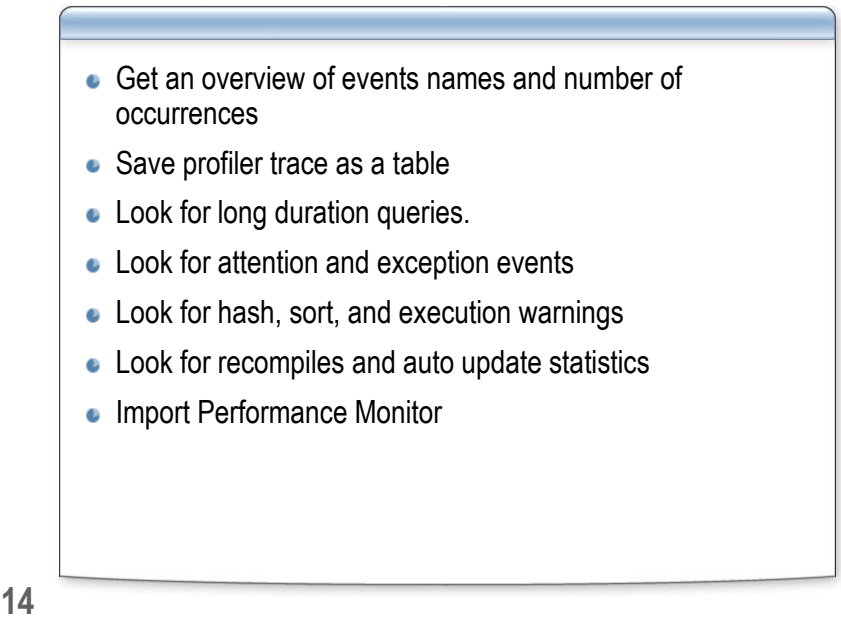
**Note:** You cannot modify the objects traced by the default trace.

You can disable the default trace by running the following command:

```
Sp_configure 'default trace enabled', 0
```

**Caution**: Turning off the default trace prevents a few SQL Server Management Studio reports, such as Configuration Changes or Schema Changes, from working correctly.

# Analyzing SQL Server Profiler Output




To determine how often an event occurs and return additional information about the event, run the following query:

```
SELECT eventclass, Count(eventclass) as CountOfEvents
FROM ::fn_trace_gettable('C:\path\trace_name.trc',default)
Group by eventclass
order by countofevents
```

The result of the query above displays each eventclass by its ID.

> **Note**: For more information about each event ID, refer to the topic *sp_trace_setevent (Transact-SQL)* in SQL Server Books Online.

## Filtering

You can use SQL Server Profiler to add filters based on criteria, such as SPID, time range, duration, CPU, reads, writes, or application name. Most filters are simple equality filters, but you can also add comparison filters and pattern matches similar to LIKE. If you already know what the problem may be based on the description of the problem, you can use these rudimentary filters to find queries that have high values for duration, reads, writes, and so on.

It is common to filter queries based on the **Duration** column. For example, you can find all queries that took longer than two seconds to run. However, it is difficult to distinguish whether this duration was associated with actual execution time or whether the query was

blocked while waiting on a lock. In this case, you can correlate the data with blocker script output to obtain more information about the cause of the problem.

### Grouping

You can also use SQL Server Profiler to group data and sort it based on the grouping column. If you group the Profiler data by duration, the events are also sorted (in an ascending order) by duration. This is a popular way to quickly find individual queries that have a high value for a particular column.

You can also group data by event class to determine how events are distributed. For example, you can group Profiler data by the event class **Attention** to check if there are many queries being cancelled by users or applications. Excessive attention events may indicate poor query performance because users grow tired of waiting for their queries to return results. If you also find excessive blocking in addition to finding a large number of attention events, then you might have an application that does not properly handle query cancellation.

Grouping by event class also enables you to easily find events that are typically problematic. For example, if you find **Missing Column Statistics** events, you can assume that the query associated with this event does not have the best query plan because columns statistics were updated as part of its optimization.

### Increasing filtering capability by using trace tables

You can increase the filtering capability of SQL Server Profiler by moving the trace output to a table by using the **::fn_trace_gettable** system function as follows:

```
SELECT * INTO trace_table
FROM ::fn_trace_gettable('c:\my_trace.trc', default)
```

Although you can run simple queries directly against a trace file by using the **::fn_trace_gettable**, it is faster to load the data directly into a temporary table. Loading the data this way is much faster than trying to load it by using the **Save As** command in SQL Profiler to export the data to a table. If you load the data to a temporary table, you do not need to wait for the GUI to load the file and you can automatically load the data from multiple rollover files in one command (and into one table).

The only limitation with querying the trace directly or saving it to a table is that the query plan for **Showplan** and **ExecutionPlan** events are stored in a binary format. The SQL Server Profiler GUI understands this data and displays it in a readable format.

The following example query identifies the batches within a trace that result in the greatest number of reads:

```
select substring(textdata,1,50), count(*) as num_recs, avg(duration) as
avg_duration, avg(cpu) as avg_cpu,
avg(reads) as avg_reads, avg(writes) as avg_writes
from trace_table
where eventclass = 12 – SQL:BatchCompleted
GROUP BY substring(textdata,1,50)
ORDER BY avg_reads DESC
```

The following example queries the trace file directly:

```
select substring(textdata,1,50), count(*) as num_recs, avg(duration) as
avg_duration, avg(cpu) as avg_cpu,
avg(reads) as avg_reads, avg(writes) as avg_writes
from ::fn_trace_gettable('C:\path\trace_name.trc',default)
where eventclass = 12 – SQL:BatchCompleted
GROUP BY substring(textdata,1,50)
ORDER BY avg_reads DESC
```

The **EventClass** column is an integer; therefore, you need to know the corresponding **EventClass** name. The event numbers are documented in the sp_trace_setevent topic in SQL Server Books Online. The events are also stored in **sys.trace_events**, so you can join the EventClass column to the **sys.trace_events.trace_event_id** column to display the **EventClass** name. The following example returns a count of the events by name:

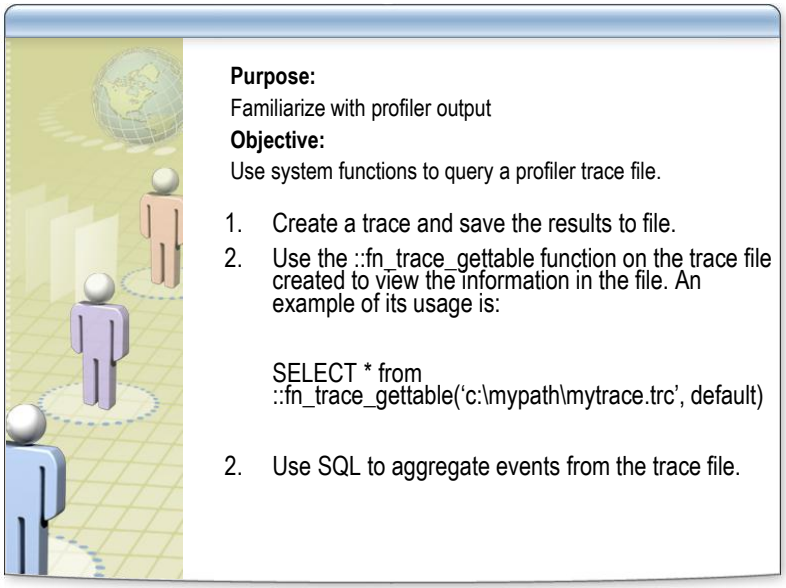
```
SELECT name, count(name) as CountOfEvents
FROM ::fn_trace_gettable('C:\path\trace_name.trc',default)
Join sys.trace_events on eventclass= trace_event_id
Group by name
order by countofevents
```

## Importing Performance Monitor (Perfmon)

If you have captured a Performance Monitor (Perfmon) log from the same time period as a Profiler trace, you can import the Perfmon log into SQL Server Profiler. Then, as you navigate through events, the corresponding time in the Perfmon log is displayed, and vice versa. To import a Perfmon log into SQL Server Profiler, on the **File** menu, click **Import Performance Data**. The trace must include the **StartTime** and **EndTime** columns.

> **Note:** When producing an event, the server always produces the **Duration** column in microseconds (10E-6) instead of milliseconds (ms). When viewing a trace file, SQL Server Profiler has an option to display the duration values as ms (currently the default) similar to SQL Server 2000. If you use the GUI to save a file, it still saves the original value in microseconds. If you read a trace with **fn_trace_gettable**, it always returns the raw value stored in the file (microseconds for a SQL Server 2005 and above trace, and ms for SQL Server 2000 trace). Therefore, you must interpret the values appropriately.

# Demonstration 4: Profiler output



**Purpose:**

Familiarize with profiler output

**Objective:**

Use system functions to query a profiler trace file.

1. Create a trace and save the results to file.
2. Use the ::fn_trace_gettable function on the trace file created to view the information in the file. An example of its usage is:

```
SELECT * from
::fn_trace_gettable('c:\mypath\mytrace.trc', default)
```

2. Use SQL to aggregate events from the trace file.

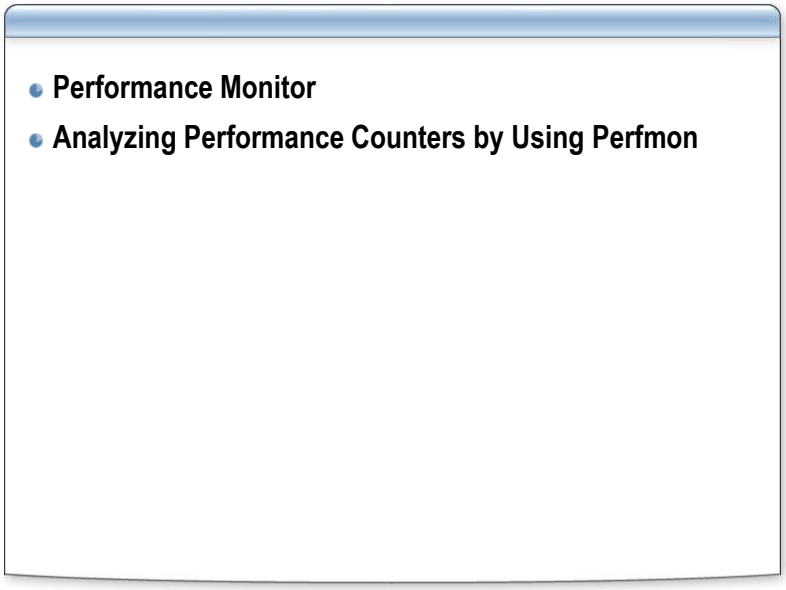
15

# Section 2 Review

- When setting up a Profiler trace, should you include all events or should you use server-side tracing?
- What columns in the trace output can you use to identify expensive queries?
- How can you use the default trace for troubleshooting performance issues?

16

# Section 3: Working with Performance Monitor




## Introduction

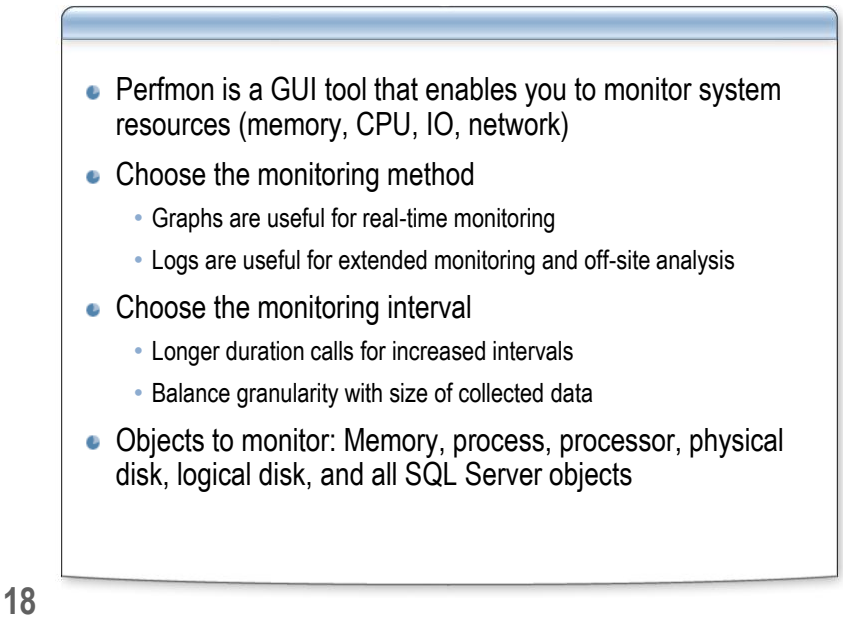
This section describes how to configure Performance Monitor (Perfmon) logging. It also explains the counters that you can use as a starting point when troubleshooting performance.

## Objectives

After completing this section, you will be able to:

- Explain the role of Performance Monitor (Perfmon) in evaluating the performance of SQL Server.
- Analyze the performance of SQL Server by monitoring system-performance objects and counters with Performance Monitor.

# Performance Monitor




The Windows Performance Monitor (also called Perfmon) is a tool for monitoring resource usage on any computer running Windows. Perfmon contains various counters, each of which measures a specific resource on the computer. You can use Perfmon to either view system-performance objects and counters in real time or log performance-counter information for later analysis.

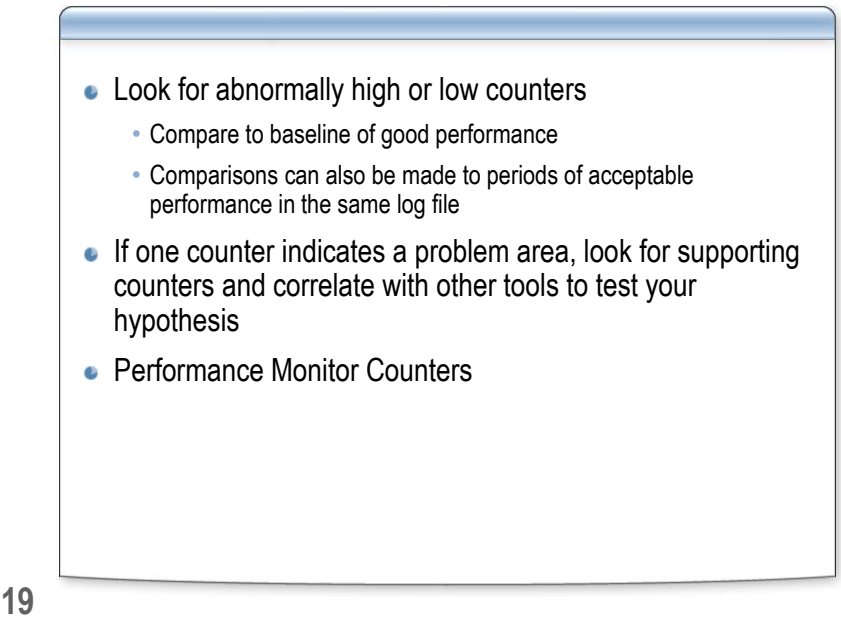
You can capture Perfmon data for a period of time and then analyze and compare log files. Logging the information provides a view of system behavior over time, which can reveal trends in system usage that could be missed when examining short periods of real-time data. You should limit the log file size to 250 MB and enable the Rollover feature.

Log files can become very large depending on the number of counters collected and the frequency of the collection interval. If this happens, consider increasing the sampling interval when logging performance data. Generally, a 30 to 60-second interval is adequate to reflect system state and trends. There are cases when you should use shorter logging periods with smaller intervals, because long sampling intervals can miss peaks and valleys in the data.

The longer the period of time for which you want to collect data, the longer you should make the sampling interval. For instance, if you want to collect detailed data for 30 minutes, use a smaller interval, such as 10 or 15 seconds. If you want to collect data for an entire 24-hour period, then 60 seconds might be an appropriate interval to keep the log file size manageable.

**Note:** When you use Perfmon to log data, you should not log the data over a network connection. It is more efficient to collect the log locally on the server being monitored. If you must log remotely over the network, you should reduce the number of objects and counters to only those that are most crucial.

# Analyzing Performance Counters by Using Perfmon




When trying to isolate a performance problem, you should look for extreme values in the performance counters. If a given counter typically averages five and it suddenly changes to a significantly lower or higher value, then note this change and consider its effect on performance. It may or may not be related to the performance problem. If benchmarks are established, you should compare counters to the benchmark.

If the value of one counter changes significantly, it is often helpful to look for other counters that could also be affected by the problem. You should determine if these counters changed significantly during the same time.

## Memory

Memory should be the first counter that you analyze. If the server is running low on memory, CPU and I/O usage can increase.

Paging is not necessarily a bad thing. Paging is only bad when a critical process must wait for its pages to be ***in-paged***, or when the amount of read/write paging is causing either excessive kernel time or disk I/O, and interfering with the normal user-mode processing.

## Available Bytes

When evaluating the Available Bytes counter, you must consider the following:

- Memory cannot be added as a solution. Further investigation is necessary to determine what is consuming physical memory. Adding more memory may alleviate the problem but does not address the root cause.

- Remember that extensive paging will occur at this point and the system will slow down. Paging is the result of this activity, not the cause.
- Monitor each process and find out which process has taken a lot of physical memory (Working Set and not Private Bytes of Individual Process). If SQL Server is configured to use Address Windowing Extensions, AWE, these counters are invalid. For SQL Server, it is always best to use the SQL Memory Manager:TotalPages and SQL Memory Manager:Target Pages counters.
- An ideal number is to have Available Bytes greater than 100 MB. If it drops below, you can feel the performance impact of not having enough memory for SQL Server, because you may see excessive page faults per second, which indicates a physical memory bottleneck. These page faults take time and CPU effort to reload pages back into physical memory.
- High CPU utilization and higher-than-normal Disk Queue Lengths will be seen under severe physical memory pressure (typically, when Available Bytes is less than 5 MB).

Generally, 64-bit versions of the operating system require more memory than 32-bit versions. Therefore, when a server has more than 4 gigabytes (GB) of memory, we recommend setting the max server memory configuration in SQL Server. The value of the max server memory should be set depending on how much physical memory is on the server and what other applications are running on the server. SQL Agent, replication jobs, and SQL Server Integration Service (SSIS) packages are all considered other applications. So, even on a dedicated SQL Server machine, you will have other executables running on the server. In general, consider leaving enough available memory for applications and the operating system based on the table below:

| Physical Memory | Available Memory under heavy load |
|---|---|
| < 4 GB | 512 MB – 1 GB |
| 4 – 32 GB | 1 – 2 GB |
| 32 – 128 GB | 2 – 4 GB |
| 128 GB | 4 GB |

**Note**: For more information, refer to http://blogs.msdn.com/slavao/archive/2006/11/13/q-a-does-sql-server-always-respond-to-memory-pressure.aspx.

## Pages/Sec and Page Faults/Sec

Pages/sec indicates hard faults, which means that physical disk I/O was incurred. Pages/sec is the rate at which pages are read from or written to disk to resolve hard page faults. This counter is a primary indicator of the kinds of faults that cause system-wide delays. Investigate if there are over 100 Pages/sec on a system with a slow disk. Usually, even 500 pages per second on a system with a fast disk subsystem may not be an issue.

Page Faults/sec indicates soft page faults.

> **Note**: For more information, refer to *How to determine the appropriate page file size for 64-bit versions of Windows Server 2003 or Windows XP* at http://support.microsoft.com/kb/889654/en-us.

## CPU

Compare system processor time to similar counters for the sqlservr instance of the Process object.

If %Processor Time is more than 85 percent on a sustained basis, you should investigate how SQL Server is using the CPU.

- **Processor: %Processor Time**

  This counter can be reported for either each logical processor on the server or a total, which is an average across all logical processors. You want to look at all the processors if affinity mask is not enabled and make sure there is an even distribution of CPU usage across all processors.

  The goal is to keep the average %Processor Time for all processors less than 80 percent. It is acceptable, however, to have brief spikes of 100 percent when SQL Server is very busy handling processor-intensive queries. You should not be too worried unless these spikes in CPU usage last much longer than any specific query. This will prevent you from thinking that you need to buy faster CPUs just because the server processed a large hash join or some other CPU-intensive activity.

  Hardware itself is probably not the bottleneck, unless you find that the CPU is over 80 percent for several hours out of the normal workday. And even then, you need to be careful to make sure that the CPUs are being utilized for efficient work instead of inefficient work (such as recompiles, unnecessary hash joins, etc.).

- **Process: %Processor Time**

  This counter can be reported for either each process on the server or a total, which is an average across all processes. It will show you the amount of CPU time used by all threads running as part of the selected process. Typically, you will be concerned with the %Processor Time for the sqlservr instance.

  Note that this counter is not normalized to the number of processors in use by Windows, so a multithreaded application, such as SQL Server, could theoretically use up to 200 percent processor time on a two-processor server, 400 percent on a four-processor server, etc. If the normalized value is much lower than the Total %Processor Time for the server, then you will need to examine %Processor Time for all the other process instances to determine which non-SQL Server process is consuming CPU time.

- **Kernel CPU versus User CPU**

  If the ratio of Kernel CPU (%Privileged time) divided by User CPU (%User Time) is greater than .25, this may indicate a network, disk, or other hardware issue. Network and Disk I/O are serviced in kernel mode. SQL Server is serviced in user mode. You can look at the Process counters for individual processes to determine which processes are consuming large amounts of kernel processing time.

## Disk

In some cases you may need to look at both physical disk and logical disk counters.

Example – If a single disk has multiple partitions then there will be multiple instances of logical disk counters and one physical disk counter. Disk 0 has 3 partitions E:, F: and G: and we have SQL data on E:, log on F: and tempdb on G:. In this case, you can see how much I/O is going to each set of partition by monitoring the logical disk counters and not physical.

Transfer time for disk reads/writes should be less than 10 ms (check with storage hardware vendor) with no queuing. If transfer time is higher than expected, check to see how SQL Server is using I/O resources.

- **Avg Disk Sec/Transfer**

  Avg Disk Sec/Transfer is the time in seconds of the average disk transfer. This is perhaps the most important counter because it is what the application actually sees. A high value for this counter should be a concern even if the disk queuing counters do not appear to show a problem.

  There are also counters—Avg Disk Sec/Read and Avg Disk Sec/Write—that help determine if a problem is primarily with reading or writing. Note that caching controllers require evaluation of reads separately from writes.

  Recommended disk transfer times:

  - Cache < 5 ms
  - SAN Fabric < 3-6 ms

  For traditional disk controllers:

  - Excellent < 15 ms
  - Good < 30 ms
  - Fair < 60 ms
  - Poor < 90 ms

  If the Avg Disk Sec/Transfer is high, you must analyze Avg Disk Queue Length and Current Queue Length. Understanding queue lengths requires an understanding of how many drives are presented to the operating system as a single disk. Additionally, verify that transaction logs are not on the same drives as the database files.

- **Avg Disk Queue Length**

  Avg Disk Queue Length is the average number of both read and write requests that queued for the selected logical disk during the sample interval. There are also separate counters for showing the queue length for reads and writes individually. For Avg Disk Queue Length:

  - Less than two plus the number of spindles is an excellent value.
  - Less than double the number of spindles is a fair value.

  This requires further investigation of the disk transfer time in order to see whether disk queue length would actually affect the system.

  Storage Area Network (SAN) architecture recommend using their data-collection tools rather than relying on the accuracy of these counters.

## SQL Server counters

### Access Methods

This object provides the following counters to monitor how the logical data within a database is accessed:

| Counter | Description |
|---|---|
| FreeSpace Scans/sec | Specifies the number of scans per second that were initiated to search for free space within pages already allocated to an allocation unit to insert or modify record fragment. Each scan may find multiple pages. High values for this counter may indicate a lot of tables with no clustered indexes (heap). If a heap is large, the insert performance may suffer. Generally, you want this value to be less than 10 per 100 Batch Requests/Sec. |
| Page Splits/sec | Specifies the number of page splits per second that occur as the result of overflowing index pages. Page splits occur on tables with clustered indexes. If the counter is high, consider using Fill Factors when creating clustered indexes. Generally, you want this value to be less than 20 per 100 Batch Requests/Sec. |
| Workfiles Created/sec | Specifies the number of work files created per second. For example, work files could be used to store temporary results for hash joins and hash aggregates. High values can increase tempdb usage and may cause I/O bottlenecks. So, if the counters are high, you should analyze Logical Disk:Avg Disk sec/transfer. Generally, you want this value to be less than 20 per 100 Batch Requests/Sec. |
| Worktables Created/sec | Specifies the number of work tables created per second. For example, work tables could be used to store temporary results for query spool, lob variables, XML variables, and cursors. High values can increase tempdb usage and may cause I/O bottlenecks. So, if the counters are high, you should analyze Logical Disk:Avg Disk sec/transfer. Generally, you want this value to be less than 20 per 100 Batch Requests/Sec. |

### Buffer Manager

This object provides counters to monitor how SQL Server uses memory. Monitoring the memory and the counters used by SQL Server helps you determine:

- Memory used to store data pages, internal data structures, and the procedure cache.

- Counters to monitor the physical I/O as SQL Server reads and writes database pages.
- If there are bottlenecks as a result of inadequate physical memory. If SQL Server cannot store frequently accessed data in cache, SQL Server must retrieve the data from disk.
- If query performance can be improved by adding more memory, or by making more memory available to the data cache or SQL Server internal structures.
- How often SQL Server needs to read data from disk. Compared with other operations, such as memory access, physical I/O consumes a lot of time. Minimizing physical I/O can improve query performance.

The following table describes the various counters provided by the Buffer Manager object:

| Counter | Description |
|---|---|
| Buffer cache hit ratio | Specifies the percentage of pages found in the buffer cache without having to read from disk. The ratio is the total number of cache hits divided by the total number of cache lookups over the last few thousand page accesses. After a long period of time, the ratio moves very little. Because reading from the cache is much less expensive than reading from disk, you want this ratio to be high. Generally, you can increase the buffer cache hit ratio by increasing the amount of memory available to SQL Server. Generally, you want the value of the Buffer cache hit ratio counter to be greater than 90 percent. |
| Checkpoint pages/sec | Specifies the number of pages flushed to disk per second by a checkpoint or other operation that requires all dirty pages to be flushed. |
| Database pages | Specifies the number of pages in the buffer pool with database content. You should compare the value of this counter to Total pages. Ideally, you want Database pages to be a large percent of the Total pages. |
| Free list stalls/sec | Specifies the number of requests per second that had to wait for a free page. |
| Free pages | Specifies the total number of pages on all free lists. You should compare the value of this counter to Total pages. If Free pages becomes low, you should check if Lazy writes/sec increases. If Free pages is low, that may indicate memory pressure inside SQL Server. You should analyze Memory:Available Mbytes to check if there is memory pressure inside the server. Generally, you want the value of the Free pages counter to be greater than 640. |
| Lazy writes/sec | Specifies the number of buffers written per second by the lazy writer of the buffer manager. The lazy writer is a system process that flushes out batches of dirty, aged buffers and makes them available to user processes. Aged buffers are buffers that contain changes that must be written back to disk before the buffer can be reused for a different page. The lazy writer eliminates the need to perform frequent checkpoints in order to create available buffers. High values of the Lazy writes/sec counter can indicate memory pressure inside SQL Server. When this counter increases, you should check if the values for Free pages and Page life expectancy counters decrease. Generally, you do not want the value of Lazy writes/sec greater than 0 for extended periods. |
| Page life expectancy | Specifies the number of seconds for which a page will stay in the buffer pool without references. Low values of this counter may indicate memory pressure or queries that perform a lot of reads. When the value of this counter decreases, you should check if the value for Lazy writes/sec is increasing and Free pages is decreasing. Generally, you want the value of the Page life expectancy counter |

| Counter | Description |
|---|---|
| | to be greater than 300. |
| Stolen pages | Specifies the number of pages used for miscellaneous server purposes (including procedure cache). You should compare the value of this counter to that of Total pages. A high percentage of Stolen pages may cause memory pressure inside SQL Server. |
| Target pages | Specifies the ideal number of pages in the buffer pool. |
| Total pages | Specifies the number of pages in the buffer pool (including database, free, and stolen pages). |

### Cursors

The following table describes the various counters provided by the Cursors object:

| Counter | Description |
|---|---|
| Cached Cursor Counts | Specifies the number of cursors of a given type in the cache. |
| Cursor memory usage | Specifies the amount of memory consumed by cursors in kilobytes (KB). |
| Cursor Requests/sec | Specifies the number of SQL cursor requests received by server. High values may indicate an inefficient use of cursors. |
| Cursor worktable usage | Specifies the number of worktables used by cursors. High value may induce stress in tempdb and may cause I/O bottlenecks. |

### SQL Server:General Statistics

The following table describes the various counters provided by the SQL Server:General Statistics object:

| Counter | Description |
|---|---|
| Active temp tables | Specifies the number of temporary tables or table variables in use. |
| Temp tables creation rate | Specifies the number of temporary tables or table variables created per second. |
| Temp tables for destruction | Specifies the number of temporary tables or table variables waiting to be destroyed by the cleanup system thread. |
| Processes Blocked | Specifies the number of currently blocked processes. |

### SQL Server:Latches

The following table describes the various counters provided by the SQL Server:Latches object:

| Counter | Description |
|---|---|
| Latches | Are the internal SQL Server resource locks that are taken when moving a page from disk to memory or vice versa. High values for this counter may indicate an I/O bottleneck. You should analyze Logical Disk:Avg Disk sec/transfer to check if the I/O performance is acceptable. |
| Average Latch Wait Time (ms) | Specifies the average latch wait time (in ms) for latch requests that had to wait. |
| Latch Waits/sec | Specifies the number of latch requests that could not be granted immediately. |

| Counter | Description |
|---|---|
| Total Latch Wait Time (ms) | Specifies the total latch wait time (in ms) for latch requests in the last second. |

### SQL Server:Locks

The following table describes the various counters provided by the SQL Server: Locks object:

| Counter | Description |
|---|---|
| Average Wait Time (ms) | Specifies the average amount of wait time (in ms) for each lock request that resulted in a wait. |
| Lock Timeouts (timeout > 0)/sec | Specifies the number of lock requests per second that timed out, excluding the requests for NOWAIT locks. |
| Lock Waits/sec | Specifies the number of lock requests per second that required the caller to wait. |
| Number of Deadlocks/sec | Specifies the number of lock requests per second that resulted in a deadlock. |

High values for Average Wait Time (ms), Lock Waits/sec, and Lock Timeouts (timeout > 0)/sec may indicate blocking. You should consider using the Blocked Process Report event in SQL Server Profiler to determine if blocking occurs.

### SQL Server:Memory Manager

Compare individual categories (Lock Memory, Optimizer Memory, etc.) to Target Server Memory.

### Memory Grants Pending

This counter specifies the total number of processes waiting for a workspace memory grant. The counter records the number of SPIDs that are waiting to acquire memory.

### Plan Cache

The Plan Cache object provides counters to monitor how Microsoft SQL Server uses memory to store objects, such as stored procedures, ad hoc and prepared Transact-SQL (T-SQL) statements, and triggers. The following table describes the various types of plans in the Plan Cache:

| Instance | Description |
|---|---|
| _Total | Specifies the cumulative for all plan types. |
| SQL Plans | Specifies the query plan from ad hoc T-SQL statements. |
| Object Plans | Specifies the query plans for stored procedures, functions, and triggers. |
| Bound Trees | Specifies the normalized trees for views, rules, computed columns, and check constraints. |
| Extended Stored Procedures | Is related to extended stored procedures. |
| Temporary Tables and Table variables | Specifies the cache related to table variables and temporary tables. |

Comparing the Cache Pages counter for each instance to the _Total counter helps determine the utilization percentage of the total Plan Cache by each instance. For example, if you compare the number of Cache Pages for the SQL Plans instance to the _Total counter and determine that most of the plan cache is for SQL Plans, it may indicate that the application issues a lot of ad hoc SQL statements and that they may not be auto parameterized.

### SQL Statistics

The following table describes the various counters provided by the SQL Statistics object:

| Counter | Description |
|---|---|
| Batch Requests/sec | Specifies the number of T-SQL command batches received per second. This statistic is affected by all constraints (such as I/O, number of users, cache size, complexity of requests, etc.). High batch requests mean good throughput. |
| SQL Attention rate | Specifies the number of attentions per second. An attention is a request by the client to end the currently running request. High values for this counter may indicate blocking and queries reaching the query timeout threshold. |
| SQL Compilations/sec | Specifies the number of SQL compilations per second. This counter indicates the number of times the compile code path is entered. The value for this counter also includes the compiles caused by statement-level recompilations in SQL Server 2005. After SQL Server user activity is stable, this value reaches a steady state. Compilation indicates that no plan was found in cache. High values for this counter may indicate that the application could be more efficient if it used stored procedures or parameterized queries by using sp_executesql. Because compilation tends to be a CPU-intensive operation, you should note the value of this counter when prolonged high CPU is observed. Generally, you want this value to be less than 10 percent of the number of Batch Requests/sec. |
| SQL Re-Compilations/sec | Specifies the number of statement recompiles per second. The counter counts the number of times statement recompiles are triggered. Generally, you want the recompiles to be low. In SQL Server 2005, recompilations are statement-scoped instead of batch-scoped. Therefore, direct comparison of values of this counter between SQL Server 2005 and earlier versions is not possible. Recompiles can occur due to changes in SET options, owner qualification, schema changes on the underlying object, references to temporary tables created in different procedures, etc. Similar to compilation, recompilation is also CPU intensive. Generally, you want the value of the SQL Re-Compilations/sec counter to be less than 10 percent of the number of Batch Requests/sec, indicating that whatever plans are being found in the cache can be reused without going through the original work of recompiling them. Frequent recompiles negate the advantage of using precompiled stored procedures. |

#### Batch Requests/sec versus SQL Compilations/sec

The worst case is when compilations are very high as compared to batch requests. This could be due to possible memory pressure in which query plans are discarded quickly to make room for other activity. Another possibility is the lack of parameterization. Parameterization is the process where variables are used instead of literal values and is important for plan reuse. In some cases, the use of sp_executeSQL can be beneficial.

> **Note**: There will be more on parameterization and plan reuse in the later modules of this workshop.

### SQL Compilations/sec versus SQL Re-compilations/sec

SQL Compilations/sec includes the initial compile of the stored procedure while SQL Re-compilations/sec only includes re-compiles. If initial compiles are low as compared to SQL re-compilations, then there is a probable re-compilation problem.

### Transactions

This object provides counters to monitor the number of transactions active in an instance of the Database Engine, and the effect of these transactions on resources, such as the snapshot isolation row version store in tempdb. When a database is set to allow snapshot isolation level, SQL Server must maintain a record of the modifications made to each row in a database. Each time a row is modified, a copy of the row as it existed before the modification is recorded in a row version store in tempdb.

You can use the following counters available in the Transaction object to monitor the version store in tempdb:

| Counter | Description |
|---|---|
| Free Space in tempdb (KB) | Specifies the amount of space (in KB) available in **tempdb**. There must be enough free space to hold both the snapshot isolation-level version store and all new temporary objects created in this instance of the Database Engine. |
| Longest Transaction Running Time | Specifies the length of time (in seconds) since the start of the transaction that has been active longer than any other current transaction. If this counter shows a very long-running transaction, you should use fn_transactions() to identify the transaction. |
| NonSnapshot Version Transactions | Specifies the number of currently active transactions that are not using snapshot isolation level and have made data modifications that have generated row versions in the tempdb version store. |
| Snapshot Transactions | Specifies the number of currently active transactions that are using the snapshot isolation level.<br>**Note**: The Snapshot Transactions counter responds when the first data access occurs, not when the BEGIN TRANSACTION statement is issued. |
| Transactions | Specifies the number of currently active transactions of all types. |
| Update conflict ratio | Specifies the percentage of those transactions using the snapshot isolation level that have encountered update conflicts within the last second. An update conflict occurs when a snapshot isolation-level transaction attempts to modify a row that was last modified by another transaction that was not committed when the snapshot isolation level transaction started. |
| Update Snapshot Transactions | Specifies the number of currently active transactions that are using the snapshot isolation level and have modified data. |
| Version Cleanup rate (KB/s) | Specifies the rate (in KB per second) at which row versions are removed from the snapshot isolation version store in tempdb. |

| Counter | Description |
|---|---|
| Version Generation rate (KB/s) | Specifies the rate (in KB per second) at which new row versions are added to the snapshot isolation version store in tempdb. |
| Version Store Size (KB) | Specifies the amount of space (in KB) in tempdb being used to store snapshot isolation level row versions. |
| Version Store unit count | Specifies the number of active allocation units in the snapshot isolation version store in tempdb. |
| Version Store unit creation | Specifies the number of allocation units that have been created in the snapshot isolation store since the instance of the Database Engine was started. |
| Version Store unit truncation | Specifies the number of allocation units that have been removed from the snapshot isolation store since the instance of the Database Engine was started. |

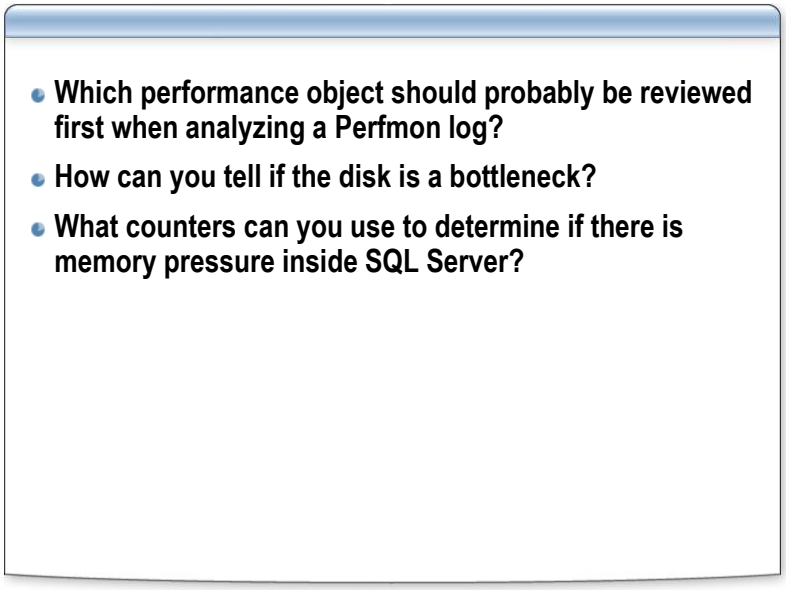
### Wait Statistics

SQL Server 2005 introduced new performance counters to track performance at a more granular level. One of those counter objects is the **Wait Statistics** object. The following table describes the counters available in the Wait Statistics object that help determine which category connections required the most wait time:

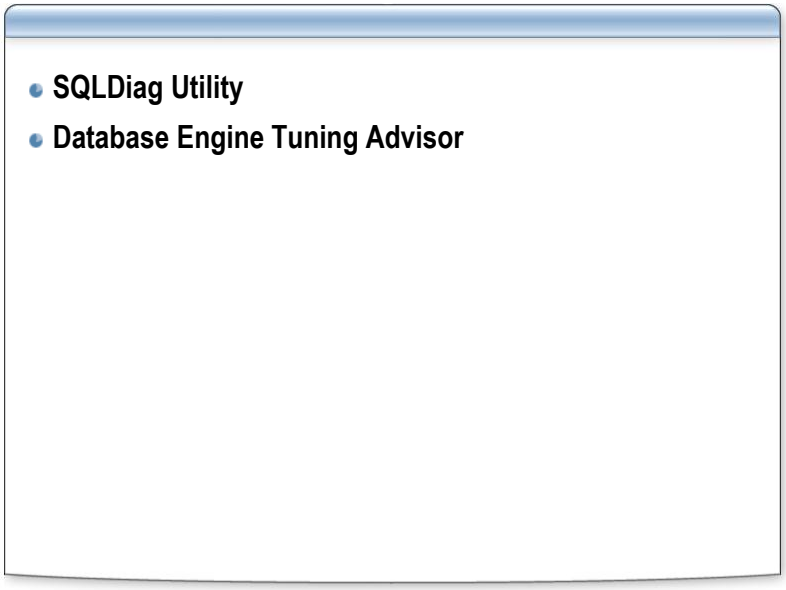
| Counter | Description |
|---|---|
| Lock Waits | Specifies the statistics for processes waiting on a lock. |
| Log Write waits | Specifies the statistics for processes waiting for log buffer to be written. |
| Memory grant queue waits | Specifies the statistics for processes waiting for memory grant to become available. |
| Page I/O latch waits | Specifies the statistics relevant to page I/O latches. |
| Page latch waits | Specifies the statistics relevant to page latches, not including I/O latches. |
| Wait for the worker | Specifies the statistics relevant to processes waiting for worker to become available. |

Some of the statistics for these counters include average wait time in ms that SQL Server has waited on the respective resources, the number of waits currently in progress, and the total number of waits progressed per second on the particular waittype.

# Section 3 Review



# Section 4: Other Performance Monitoring Tools




## Introduction

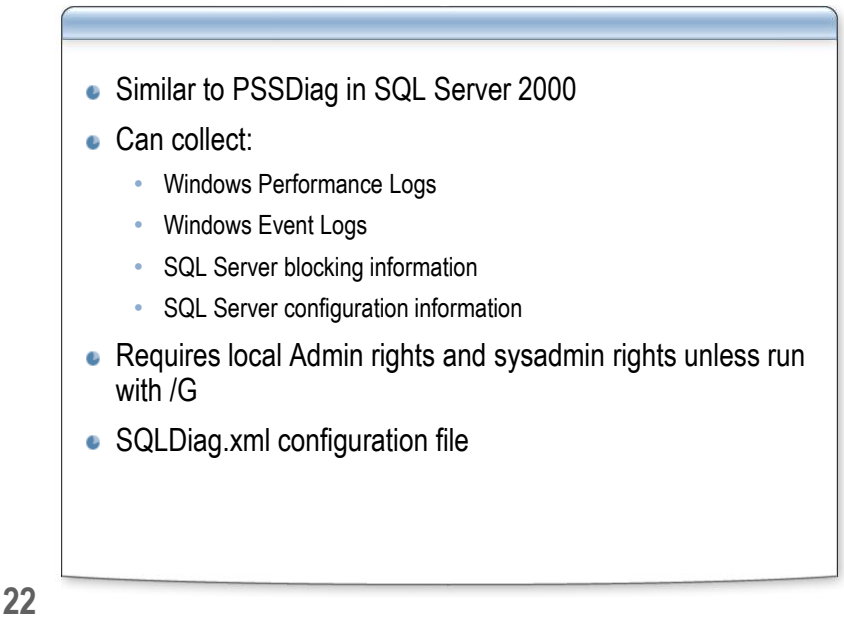
This section explains how to use SQLDiag to monitor a SQL Server and how to analyze the data captured by SQLDiag. The section also describes how to leverage Database Engine Tuning Advisor (DTA) to tune a workload.

## Objectives

After completing this section, you will be able to:

- Collect performance data by using the SQLDiag utility.
- Analyze a workload and the physical implementation of a database by using the SQL Server Database Engine Tuning Advisor (DTA).

# SQLDiag Utility




**SQLDiag** is a diagnostics collection utility that you can run either from the command prompt or as a service. This tool is similar to **PSSdiag** available in SQL Server 2000. This tool can collect the following information:

- Windows performance logs. Disabled by default; SQL counters do appear in the results set, but a separate performance monitor log is not generated.
- Windows event logs. Disabled by default.
- SQL Server Profiler traces. Disabled by default.
- SQL Server blocking information. Disabled by default.
- SQL Server configuration information.

The following table describes the command-line arguments that you can use with SQLDiag. However, note that these arguments are not compatible with SQL Server 2000.

| Argument | Description |
|---|---|
| /? | Displays the usage information. |
| /I configuration_file | Sets the configuration file for SQLDiag to use. By default, /I is set to SQLDiag.xml. |
| /O output_folder_path | Redirects the **SQLDiag** output to the specified folder. |
| /P support_folder_path | Sets the support folder path. |
| /N output_folder_management_option | Specifies whether **SQLDiag** overwrites or renames the output folder when it starts up. |
| /C | Sets the type of file compression used on the **SQLDiag** output folder |

| Argument | Description |
|---|---|
| file_compression_type | files. |
| /B [+]start_time | Specifies the date and time to start collecting diagnostic data. |
| /E [+]stop_time | Specifies the date and time to stop collecting diagnostic data. |
| /Q | Runs **SQLDiag** in quiet mode. |
| /G | Runs **SQLDiag** in generic mode. |
| /R | Registers **SQLDiag** as a service. |
| /U | Unregisters **SQLDiag** as a service. |
| /L | Runs **SQLDiag** in continuous mode. |
| /X | Runs **SQLDiag** in snapshot mode. |

**Note**: For more information about SQLDiag command-line arguments, refer to SQL Server 2005 Books Online.

## Configuration file

SQLDiag collects information based on the arguments that you specify and the configuration file **SQLDiag.xml**, which is located in **\Program Files\Microsoft SQL Server\90\Tools\Binn**. The configuration file uses the XML schema, **SQLDiag_schema.xsd**. SQL Server also includes two additional configuration files—**sd_general.xml** and **sd_detailed.xml**. These files are created after the initial run of **SQLDiag**. These configuration files have Perfmon, NT Events logs, and SQL Server Profiler enabled. The differences between the various configuration files are listed in the table below.

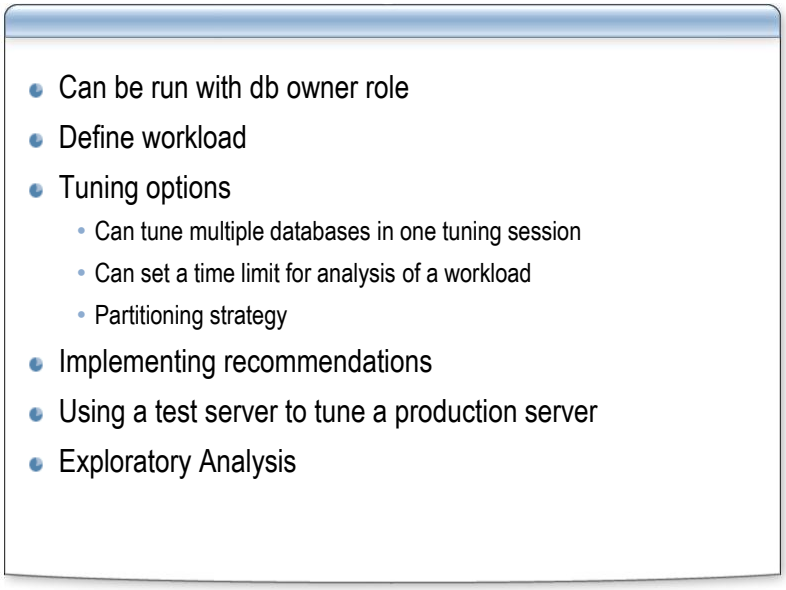
| Configuration File | Event Logs | Performance Logs | Msinfo | SQL Server Configuration | Profiler Trace | Blocking |
|---|---|---|---|---|---|---|
| SQLDiag.xml | No | No | Yes | Yes | No | No |
| sd_general.xml | Yes | Yes | Yes | Yes | Yes | No |
| sd_detailed.xml | Yes | Yes | Yes | Yes | Yes | No |

The following code snippet runs **SQLDiag** with a specific end time and sets the configuration file to sd_general.xml:

```
sqldiag /I sd_general.xml /E +00:10:00
```

If you want to customize the configuration file, you should first make a copy and then make the appropriate changes. This prevents SQLDiag from overwriting your changes the next time it runs. Also, the default configuration file, **sqldiag.xml**, is configured to shut down immediately. So, if you want to use a custom configuration, you should start with the **sd_general.xml** file.

# Database Engine Tuning Advisor




The SQL Server Database Engine Tuning Advisor (DTA) analyzes a workload and the physical implementation of one or more databases. It can then recommend that you add, remove, or modify physical design structures in your databases. In addition, it recommends the statistics that you should collect to back up these physical design structures.

## Permissions

DTA no longer requires the *system administrator* right in SQL Server. Users with the **db_owner** role can run DTA. However, after you install SQL Server, a user who is a member of the **sysadmin** fixed server role must launch DTA before anyone else can use it.

## Defining workload

The input for DTA can be a SQL Server Profiler trace, SQL script, XML file, or a trace table. If you are using a trace table, the table cannot reside on a remote server. The DTA recommendations are only as good as the workload. So, the more representative the workload is of the production queries, the better are the recommendations.

## Tuning options

There are a number of options that you can set in DTA:

- Set a session name. DTA enables you to view previous run sessions. Therefore, you should choose a session name which is descriptive.
- Specify the workload and set the database for workload analysis.

- Set **Limit Running Time** to either reduce the server load or make sure that analysis finishes within a maintenance window.
- Specify the Physical Design Structures (PDS) to recommend.
- Set **Partitioning Strategy**.
- Specify the PDS to keep.
- Set the size of index, maximum columns per index, and whether to generate offline or online index creation by using **Advanced Options**.

## Implementing DTA recommendations

The DTA provides various reports that you can use to analyze the DTA recommendations. You can either implement these recommendations by using the Action menu or save them to a file.

In an ideal scenario, you should save the recommendations to a file and then review those. You should then implement these recommendations in a test environment and veriy that they help improve the performance of the system as a whole.

## Using a test server to tune production

Tuning a large workload can create significant overhead on the server that is being tuned. The overhead results from the several calls made by the DTA to the query optimizer during the tuning process. You can eliminate this overhead problem by using a test server in addition to your production server.

The traditional way to use a test server is to copy all of the data from your production server to the test server, tune the test server, and then implement the recommendations on the production server. This process eliminates the performance impact on your production server, but nevertheless, is not the optimal solution. For example, copying large amounts of data from the production server to the test server can consume substantial amounts of time and resources. In addition, test server hardware is seldom as powerful as the hardware that is deployed for production servers. The tuning process relies on the query optimizer, and the recommendations that it generates are based in part on the underlying hardware. If the test server and the production server hardware is not identical, the quality of DTA recommendations may be diminished.

To avoid these problems, DTA tunes a database on a production server by offloading most of the tuning load onto a test server. DTA does this by using the production server hardware configuration information and without actually copying the data from the production server to the test server; it only copies the metadata and the necessary statistics.

The following steps outline the process for tuning a production database on a test server:

1. Before you start, make sure that the user who wants to use the test server to tune a database on the production server exists on both servers. This requires you to create

the user and his or her login on the test server. If you are a member of the sysadmin fixed server role on both computers, this step is not necessary.

2. Tune the workload on the test server. To tune a workload on a test server, you must use an XML input file with the **dta** command-line utility. In the XML input file, specify the name of your test server with the TestServer subelement, in addition to specifying the values for the other subelements under the TuningOptions parent element.

   During the tuning process, DTA creates a shell database on the test server. To create this shell database and tune it, DTA makes calls to the production server for the following:

   - DTA imports metadata from the production database to the test server shell database. This metadata includes empty tables, indexes, views, stored procedures, triggers, etc. This makes it possible for the workload queries to run against the test server shell database.
   - DTA imports statistics from the production server so that the query optimizer can accurately optimize queries on the test server.
   - DTA imports hardware parameters specifying the number of processors and available memory from the production server to provide the query optimizer with the information that it needs to generate a query plan.

3. After DTA finishes tuning the test server shell database, it generates a tuning recommendation.

4. Apply the recommendation received from tuning the test server to the production server.

   Sample input file for DTA:

```
<?xml version="1.0" encoding="utf-16" ?>
<DTAXML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://schemas.microsoft.com/sqlserver/2004/07/dta">
 <DTAInput>
  <Server>
   <Name>MyServerName</Name>
   <Database>
    <Name>MyDatabaseName</Name>
   </Database>
  </Server>
  <Workload>
   <File>MyWorkloadScript.sql</File>
  </Workload>
  <TuningOptions>
   <TestServer>MyTestServerName</TestServer>
   <FeatureSet>IDX</FeatureSet>
   <Partitioning>NONE</Partitioning>
   <KeepExisting>NONE</KeepExisting>
  </TuningOptions>
 </DTAInput>
</DTAXML>
```

DTA schema can be found at either http://schemas.microsoft.com/sqlserver/2004/07/dta/ or C:\Program Files\Microsoft SQL Server\90\Tools\Binn\schemas\sqlserver\2003\03\dta\dtaschema.xsd.

### Exploratory analysis

You can use exploratory analysis for tuning the configurations of existing and hypothetical physical design structures, such as indexes, indexed views, and partitioning. The benefit of specifying hypothetical structures is that you can evaluate their effects on your databases without incurring the overhead of first implementing them.

You can perform exploratory analysis in the following two modes with DTA:

- In the **evaluate mode**, DTA compares the cost of the current configuration (C) with that of a user-specified configuration (U), for the same workload. C is always a real configuration because it consists of physical design structures that currently exist in the database. On the other hand, U is a configuration that consists of real and hypothetical physical design structures. If DTA reports that the cost of U is lower than the cost of C, it is likely that the physical design of U will perform better than C.
- In the **tune mode**, a database administrator already knows that a part of the database physical design should be fixed, but he or she wants DTA to recommend the best physical design structures for the rest of the configuration.

  For example, a database administrator knows that a fact table must be partitioned because it is too large. The administrator must choose between partitioning it either by month or by quarter. Either way of partitioning the table would work, but the administrator wants to choose the partitioning method that would provide the best performance for a given workload. To determine which partitioning method is best, the administrator can use DTA to tune the workload twice. First, the administrator tunes the workload by using a user-specified configuration with the table hypothetically partitioned by month. Then, the administrator tunes the workload with the table hypothetically partitioned by quarter. After the workload has been tuned with both of the hypothetical configurations, the administrator can compare the percentage of improvement to determine which partitioning method will provide the best performance.

To export tuning session results from the DTA GUI for a ***what-if*** analysis with the dta command-line utility:

1. Use the DTA GUI to tune a database. If you want to evaluate an existing tuning session, double-click it in the Session Monitor.
2. On the **File** menu, click **Export Session Results** and save it as an XML file.
3. Open the XML file created in Step 2 in your favorite XML editor or text editor, or in Microsoft SQL Server Management Studio. Scroll down to the Configuration element. Copy and paste the **Configuration** element section into an XML input file template after the **TuningOptions** element. Save this XML input file.

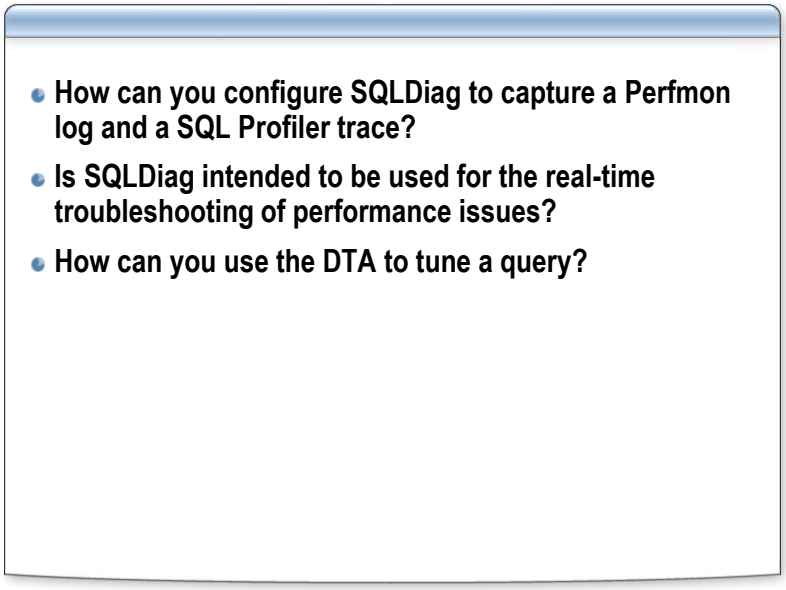
4. In the new XML input file that you created in Step 3, specify any tuning options that you want in the **TuningOptions** element, edit the Configuration element section (add or delete the physical design structures as appropriate for your analysis), save the file, and validate it against the DTA XML schema.

> **Note**: For information about editing the XML input file, refer to *XML Input File Reference (DTA)* in SQL Server Books Online.

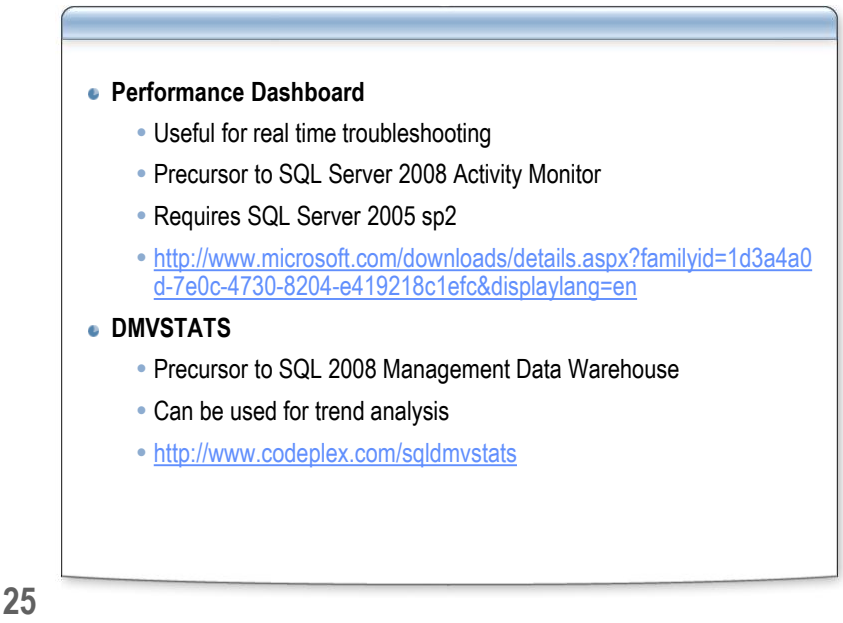
5. Use the XML file that you created in Step 4 as input to the dta command-line utility.

> **Note**: For information about using XML input files with the dta command-line utility, refer to *How to: Tune a Database by Using the dta Utility* at MShelp://MS.SQLCC.v9/MS.SQLSVR.v9.en/udb9/html/72de7bf8-ad1a-459d-b848-77b81090dcba.htm.

# Section 4 Review



# Section 5: Other Downloadable Tools




## Introduction

This section covers other downloadable tools that you can use to troubleshoot SQL Server performance. These tools are, however, not supported by Microsoft and are not shipped with SQL Server.

## Objectives

After completing this section, you will be able to identify the input required to run each tool and the location where you can download each tool from.

## Performance Dashboard

Performance Dashboard are reporting services files which use DMVs as the data source. Performance Dashboard is useful for troubleshooting real-time performance issues by using SQL Server Management Studio. After downloading this tool, run **setup.sql** script on the SQL server that you want to monitor. The SQL Server instance must be running SQL Server 2005 Service Pack 2 (SP2) or higher.

## DMVstats

DMVstats captures output from different DMVs and stores the results into tables in a DMV data warehouse.

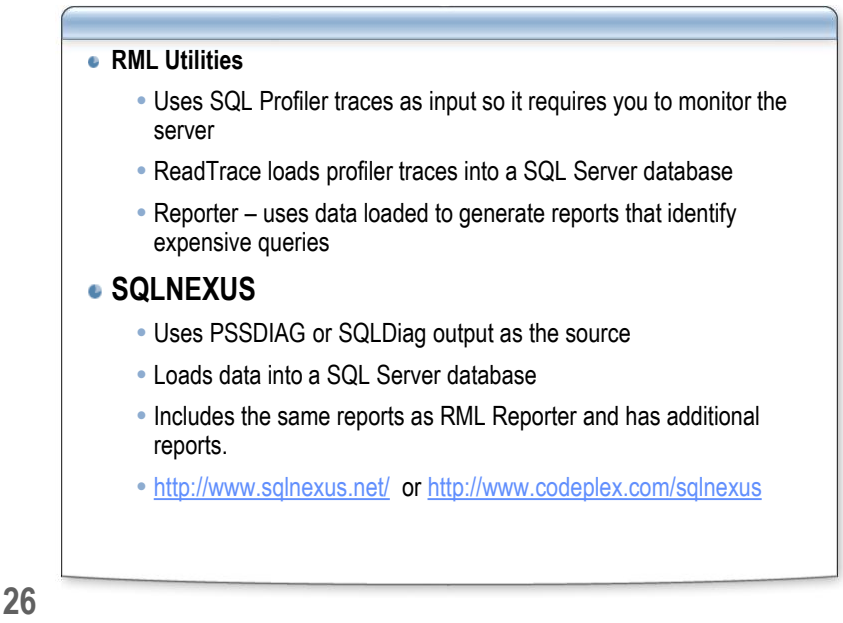
DMVstats 1.0 is an application that can collect data by using SQL Agent jobs. The application also includes some reports which you can use to analyze the data. DMVstats is for SQL Server 2005 and above. You may need to modify DMVstats for use with SQL Server 2008.

### Main components

The three main components of DMVstats are:

- DMV data collection which are implemented as SQL Agent jobs
- DMV data warehouse database
- Reports

# Other Downloadable Tools (continued)




## RML Utilities

The RML Utilities tool accepts SQL Server Profiler traces as an input and loads the trace data into a SQL Server database. Therefore, the Profiler trace will need to capture certain events in order for the data to be loaded and used in reports. Refer to the documentation to determine the necessary Profiler events. You can use the Standard template when setting a Profiler trace to produce the minimum reports.

The Profiler data is loaded into SQL Server by running ***readtrace.exe***. Again, readtrace.exe requires a connection to a SQL Server install and will create a database on the server. Therefore, you need to be able to create a database in order to run readtrace.exe. To view a complete list of parameters available for readtrace.exe, run the following command on the command prompt:

```
Readtrace /?
```

You cannot load different profiler trace data into one database. For example, if you have Profiler traces from Monday and Tuesday, you can use readtrace.exe to load the trace files from Monday into one database and those from Tuesday into another. Therefore, you cannot use RML Utilities for trend analysis.

Once RML is installed, you will see the ***RML Utilities for SQL Server*** program group. This group contains a subgroup called Reporter. Reporter is used after the data has been loaded. When you launch Reporter, you need to point it to the SQL Server and the database used when running readtrace.exe. Reporter provides different reports. The

**Unique Batch TopN's** report is used to display the most expensive queries captured by the Profiler traces.

The RML Utilities can be downloaded from:

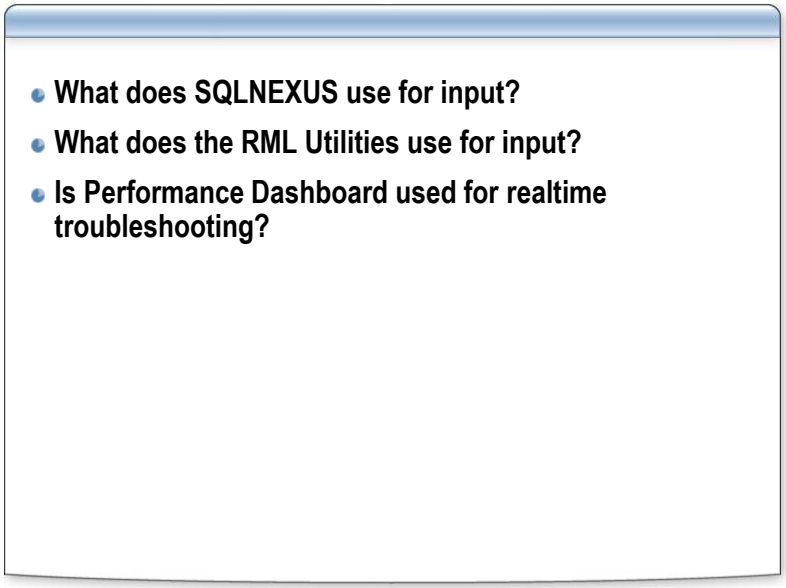
http://www.microsoft.com/downloads/details.aspx?familyid=7EDFA95A-A32F-440F-A3A8-5160C8DBE926&displaylang=en

## SQLNEXUS

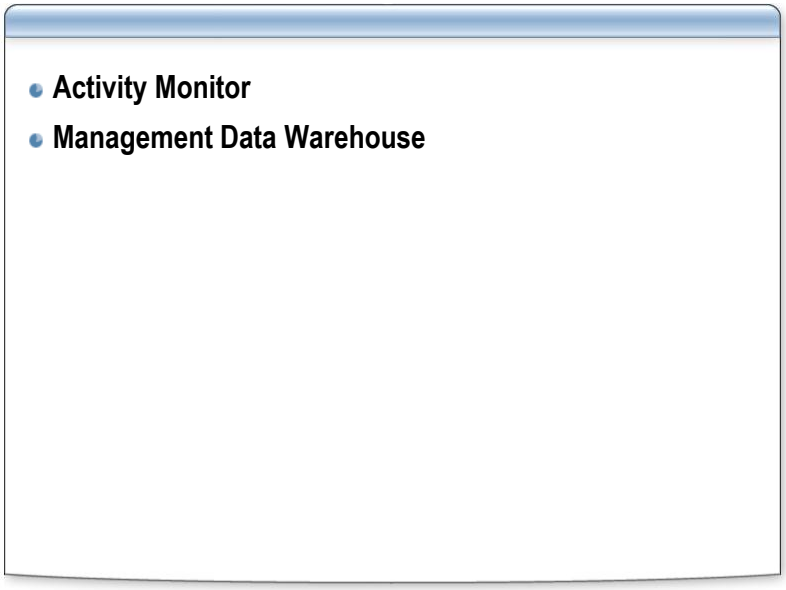
Once SQLNEXUS is installed, you will see the **SQLNEXUS** program group.

SQLNEXUS imports data captured from either PSSdiag or SQLDiag into a SQL Server database. Therefore, SQLNEXUS uses the output folder from either PSSdiag or SQLDiag as the source during the import. If PSSdiag or SQLDIAG is not configured correctly, you may either get errors when importing the data or some reports may not be available. If you are using SQLDiag as the source, all reports will not be available unless you configure SQLDiag to run the SQL Server 2005 Perf Stats Scripts. Refer to http://www.codeplex.com/sqlnexus for complete details. Additionally, the RML Utilities are a subset of SQLNEXUS. Therefore if you want to run SQLNEXUS, you need to also install the RML utilities.

# Section 5 Review



# Section 6: New Tools for SQL Server 2008




## Introduction

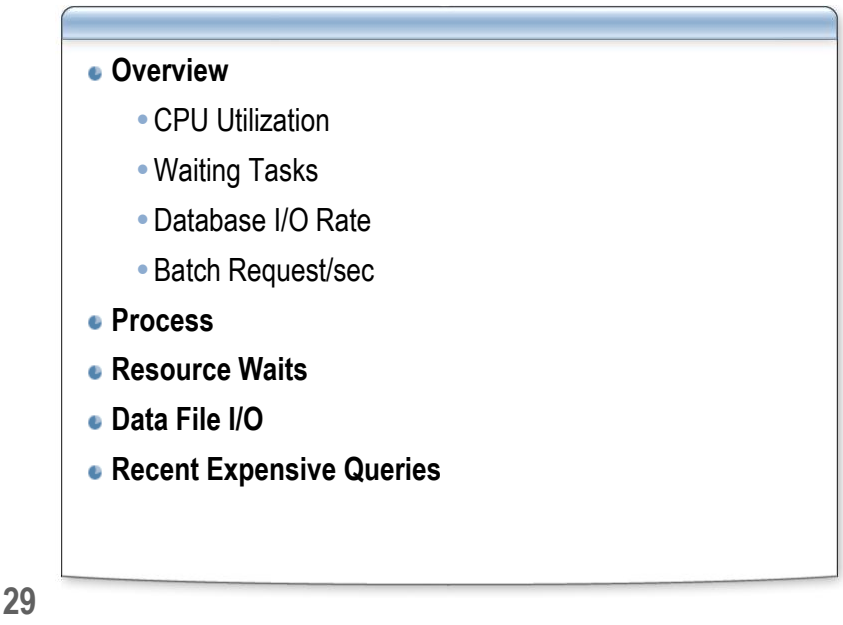
This section explains how to use Activity Monitor and Management Data Warehouse (MDW). Activity Monitor is intended for real-time troubleshooting while MDW is more useful for trend analysis.

## Objectives

After completing this section, you will be able to:

- Identify blocking and wait queues and expensive queries by using Activity Monitor.
- Identify the components of the MD architecture.
- Setup MDW to capture performance data.
- Analyze the reports produced by MDW to determine the source of bottlenecks and performance trends.

# Activity Monitor




Activity Monitor supports real-time troubleshooting of performance issues in SQL Server 2005 or higher.

Activity Monitor has a refresh interval which is configurable from one second up to one hour. The default refresh interval is 10 seconds. You can change it by right-clicking the Overview chart area. Note that using the default refresh interval in Activity Monitor has a negligible performance impact; however, shorter intervals may cause the tool to start using a few percent of system CPU. You can also pause or resume the collection of data by Activity Monitor.

Certain sections in Activity Monitor will ignore the refresh interval. The Recent Expensive Queries section ignores the default interval and refreshes once every 15 seconds, if you specify a lower refresh interval. This is by design and was done to prevent the Activity Monitor queries from causing a performance bottleneck on a large server with many requests or large procedure cache.

## CPU Utilization

This chart only shows *%Processor Time* for that SQL Server instance to which you are connected in the SQL Server Management Studio. The chart requires the user to be a member of the *Performance Monitor Users* group; otherwise, the chart will be either disabled or grayed out.

## Waiting Tasks

You can use this chart to identify wait queues. This chart shows the number of tasks (not requests) which have a waittype. By showing a count of blocked tasks and not requests,

you can view the impact of waits due to pending requests waiting for a worker thread, waits for each worker in a parallel query, etc.

For the ***Waiting Tasks*** section, Activity Monitor first groups waittypes into following categories:

- Backup: Shows waittypes associated with backups
- Buffer I/O: Shows waittypes associated with moving buffers due to I/O
- Buffer Latch: Shows waittypes associated with latches that are not a result of I/O requests
- Compilation: Shows waittypes associated with query compilation and memory grants for queries
- CPU: Shows waittypes associated with CPU and scheduler yields
- Full Text Search: Shows waittypes associated with full text
- Idle: Shows different waittypes that indicate miscellaneous wait events
- Latch: Shows waittypes associated with latches.
- Lock: Shows waittypes associated with locking
- Logging: Shows waittypes associated with the transaction log
- Memory: Shows waittypes that are associated with memory management and are waiting for memory
- Network I/O: Shows waittypes associated with communication between client and server
- Other: Shows miscellaneous waittypes
- Parallelism: Shows waittypes associated with parallel queries
- SQLCLR: Shows waittypes associated with CLR objects
- Transaction: Shows waittypes associated with T-SQL and distributed transactions
- UserWaits: Shows connections that are using the T-SQL WAITFOR statement

Activity Monitor then takes snapshots of ***sys.dm_os_wait_stats*** and ***sys.dm_os_waiting_tasks*** to produce the Waiting Tasks chart. Before the results are displayed, a weighted average formula is used to provide the user with a more stable indicator of the recent wait time. This is primarily useful if the user has selected a rapid refresh rate. With a sample interval of one second, for example, it could be hard to assess the overall bottleneck on the system if short-term fluctuations in wait time distribution were creating constant changes in the **worst** wait category for the just-completed one second sample interval.

## Database IO

This chart displays the amount of I/O, in MB/sec that the monitored SQL Server instance is generating on the system. The data for this chart comes from sys.dm_io_virtual_file_stats. This DMV simply records the number of pages that have been read or written but does not indicate whether they were random or sequential IOs.

### Batch Requests/Sec

This chart shows the number of requests/sec being processed by SQL Server.

### Processes

The Processes session in Activity Monitor shows all users tasks, including sleeping sessions.

You can set filters on the Processes session so that it displays tasks only from a particular login, application, or any other criteria that you specify. To set the filter, click the combo box in the column header and select the subset that you want returned. You can apply filter on different columns. However, in SQL Server 2008 RTM, within a column, you can apply filter based on only one value.

The head blocker column is supposed to show 1 if the session is at the head of either a blocking or a chain; otherwise, the head blocker column shows 0. In RTM, most connections have head blocker set to 1, even if a session is blocking another session. Therefore, to verify if blocking is occurring, you should check if the Blocked By column is greater than 0.

If you right-click a session, you can view the T-SQL submitted. Right-clicking a session will actually run DBCC INPUTBUFFER. You also have the option to kill the request either from here or from Trace Processes in SQL Server. The *Trace Process* option will start a SQL Profiler trace by using the SPID for a filter.

### Resource Waits

In this grid, each waittype is categorized into a wait category.

At each refresh interval and Activity Monitor saves the current total wait count and time and any pending waits into a temporary table. At the next refresh, the difference in the number of waits and the amount of wait time between the current and previous snapshot is calculated and displayed in the Resource Waits grid.

The results also include **in-progress** waits. If a session is waiting on a resource, it may not be listed in the *Recent Expensive Queries* section.

The *Recent Wait Time* column in the Resource Waits grid shows a weighted average if a short refresh interval is used.

The CPU wait category includes the waittypes SOS_SCHEDULER_YIELD and signal wait time. Therefore, it does not reflect the CPU consumed by queries. As a result, Perfmon and other tools may show high CPU usage but the CPU wait category value in Activity Monitor is low.

### Data I/O

The Data File I/O chart in Activity Monitor takes two snapshots of sys.dm_os_virtual_file_stats and inserts them into a temp table. The difference between the snapshots is calculated to produce the MB/sec. So, this chart displays the I/O rate whereas the sys.dm_os_virtual_file_stats DMV displays the total IO.

## Recent Expensive Querries

The Recent Expensive Queries section in Activity Monitor displays the top 20 expensive queries that are either currently running or have run in the last four hours. Expensive queries are based on CPU consumption rate, physical reads rate, logical writes rate, logical reads rate, and average duration. For completed queries, the section uses the sys.dm_exec_query_stats DMV. For running queries, the section uses the sys.dm_exec_requests DMV.

Activity Monitor groups the underlying queries from the DMVs by query_hash and query_plan_hash (query and plan fingerprints). This grouping helps find similar, non-parameterized queries that collectively consumed significant resources. The grouping by query_hash is not available in the performance data collector or default MDW reports. So, the MDW reports might list different queries as expensive.

## Query_hash

Query_hash is a column in sys.dm_exec_query_stats and sy.dm_exec_requests DMVs and is also called the query fingerprint. Basically, a query is hashed to a value and if another query hashes to the same value, the queries are considered to be similar and the resources being consumed by them can be added together. This is a way to group ad hoc queries that are not auto-parameterized.

Example:

```
select * from sys.objects where object_id = 100
go
select * from sys.objects where object_id = 101
go
select * from sys.objects where object_id = 102
go
```

## Query_plan_hash

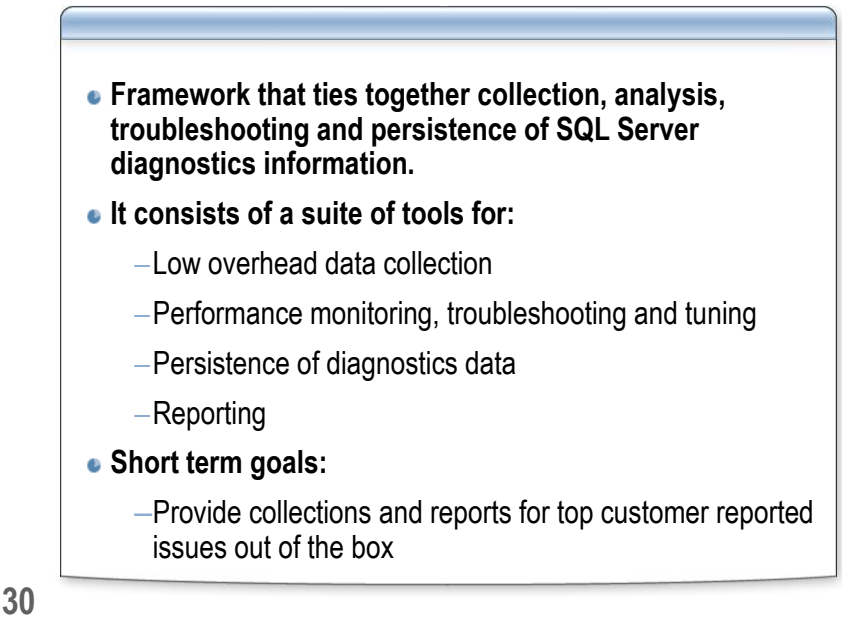
Query_plan_hash is a column in sys.dm_exec_query_stats and sy.dm_exec_requests DMVs and is also called the plan fingerprint. Basically, the query plan is hashed to a value. So, any change in the query plan hash value implies that the query plan has changed.

The recent expensive queries collector takes the delta in values (CPU consumption, reads, writes, etc.) between the current and previous collection of data X seconds ago. It then divides this difference by the refresh interval to compute a rate of resource consumption.

The Recent Expensive Queries section of the report is throttled so that it will not refresh more often than every 15 seconds. Because the values are based on a delta between snapshots, it means that under normal circumstances, you would not see any output for two refresh intervals, or 30 seconds. To avoid this big delay, when you first expand this section, it assumes a prior snapshot with zero resource consumption and the current snapshot contains the values from dm_exec_query_stats or dm_exec_requests. The section divides this delta in values by the refresh interval (for example, 15 seconds), although the values have actually been accumulating over the life of the query plan which

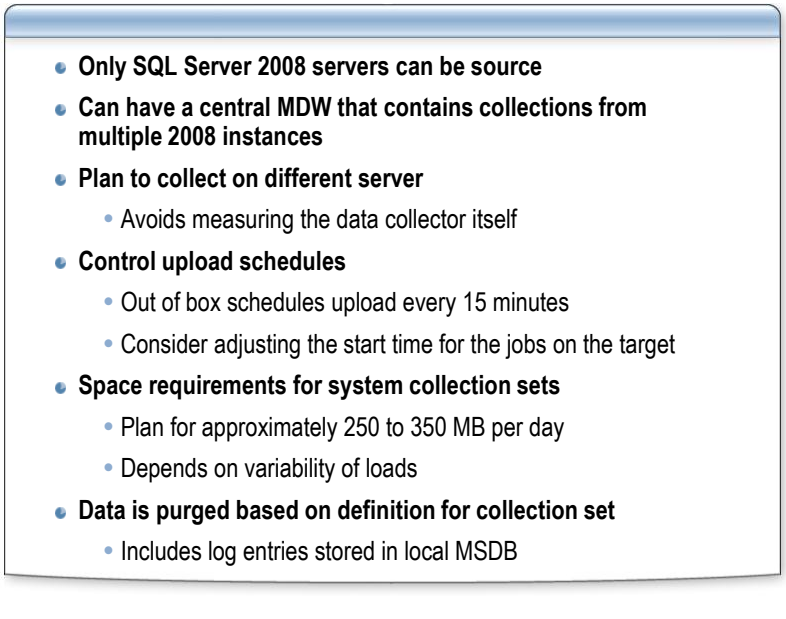
may be much longer. Consequently, the initial display of values may be either very large or impossible values for the resource consumption over a 15 second refresh interval. However, on the next refresh, the values will be calculated normally and should return to correct or expected consumption rates.

# What Is Management Data Warehouse (MDW)?




The Management Data Warehouse (MDW) is a database to store performance data collected from SQL Server instances. When MDW is enabled, SSIS packages and SQL Agent jobs are used to collect data and upload it into the MDW database. The MDW also contains several built-in reports which are used to analyze the data. The MDW is useful for historical performance and trend analysis.

# Planning for Collection

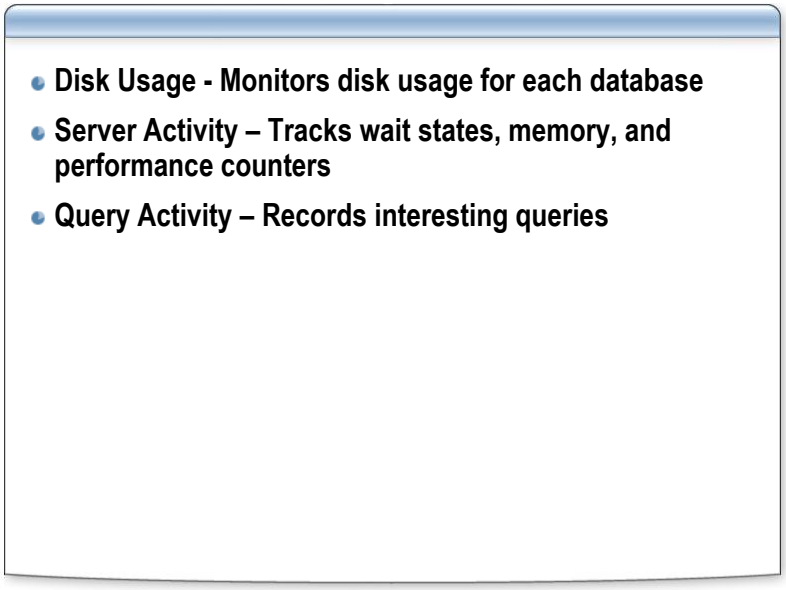



By default, there is no MDW database until you choose the *Configure Management Data Warehouse* option from within Object Explorer.

You should select the **Create or Upgrade a management data warehouse** option if you are configuring the instance to be its own repository or a central repository for other instances. If you want to monitor the instance and use that instance as its own repository, you need to run the **Configure Management Data Warehouse Wizard** and select **Setup data collection**.

When configuring multiple instances to use a single MDW, you need to launch the Configure Management Data Warehouse Wizard on each instance that you want to monitor. Next, you need to select **Setup data collection** and point it to the instance that contains the central MDW database.

# System Collection Sets

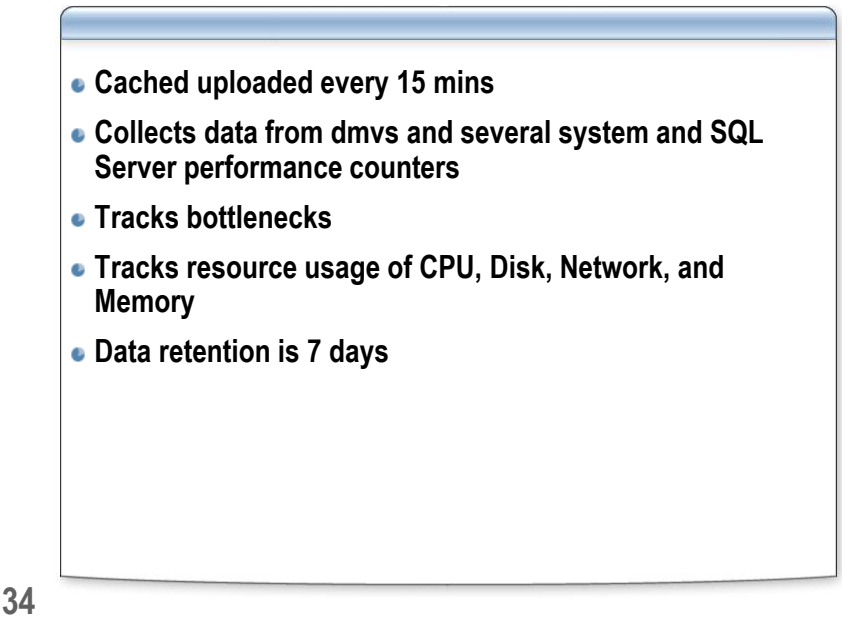


# Disk Usage Collection

- **The collection set has two collection items, Disk Usage - Data Files and Disk Usage - Log Files Both use a T-SQL Query Collector Type. The collection set gathers the following data:**
  - Snapshots of data file sizes obtained from **sys.partitions** and **sys.allocation_units**.
  - Snapshots of log file sizes obtained from **DBCC SQLPERF (LOGSPACE)**.
  - Snapshots of I/O statistics from **sys.dm_io_virtual_file_stats**.
- **Runs in non-cached mode; default schedule to upload is every 6 hrs**
  **Snapshots of I/O statistics from sys.dm_io_virtual_file_stats.**
- **Retention period is 90 days**

33

# Server Activity Collection




The Server Activity collection is comprised of four T-SQL collectors and Perfmon collectors.

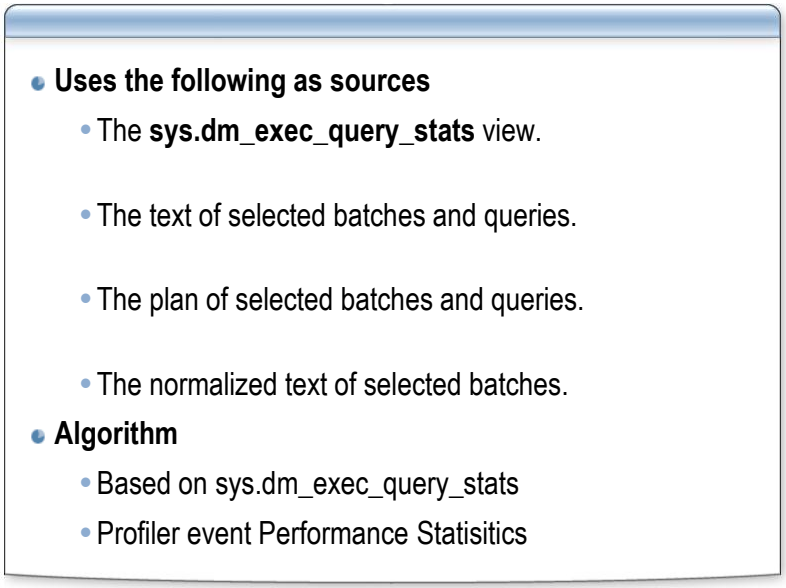
The four T-SQL collectors are:

- Server Activity – Wait Statistics
- Server Activity – Schedulers
- Server Activity – Memory
- Server Activity – Active Sessions and Requests

The Server Activity collection does not capture all Perfmon counters.

> **Note**: For the queries being issued and Perfmon counters being collected, refer to *System Data Collection Sets* at ms-help://MS.SQLCC.v10/MS.SQLSVR.v10.en/s10de_4deptrbl/html/0e49bcc7-3bab-4dd5-b5f5-62efa13864f3.htm.

# Query Statistics Collector




The algorithm to collect data is as follows:

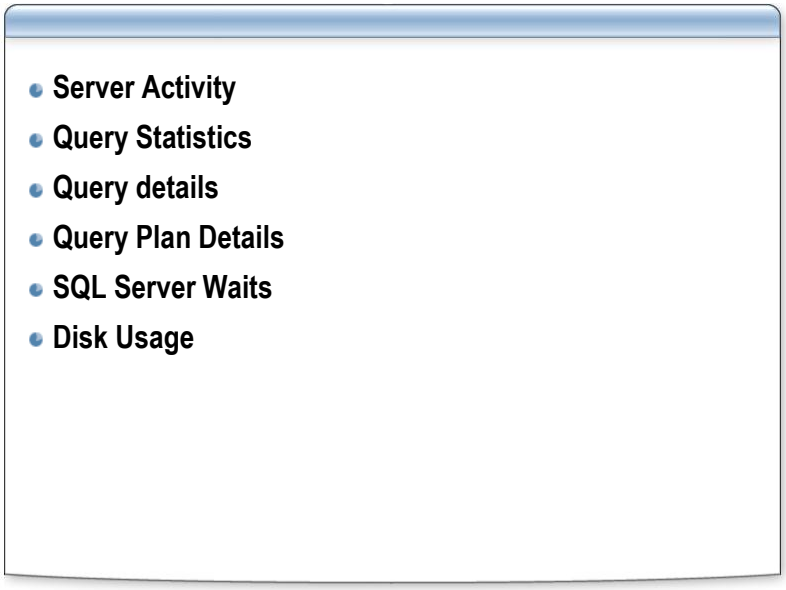
1. Collect a snapshot of **sys.dm_exec_query_stats**.
2. Retrieve the most recent snapshot (from 15 minutes earlier) for comparison with the new snapshot. The most recent snapshot is cached locally and does not have to be retrieved from the management data warehouse.
3. Select the top n queries from each snapshot by using the following metrics:
   - Elapsed time
   - Worker time
   - Logical reads
   - Logical writes
   - Physical reads
   - Execution count

   This process provides 6 x n sql_handles and plan_handles.
4. Identify the unique sql_handles and plan_handles.
5. Intersect this result with the sql_handles and plan_handles that are stored in the MDW.

6. For new sql_handles and plan_handles, get the plan and text from the server. If the plan or text cannot be found (it may have been already been removed from the local cache), store the handles in the MDW.
7. For each sql_handle text gathered, normalize the text (for example, remove parameters and literals) and calculate the unique hash value for the normalized text. Store the normalized text, the hash value, and the mapping to the original sql_handle in the MDW.

Similar to other collectors, the Query Statistics Collector also uses a SQL Profiler trace with the event Performance Statistics.

# MDW Reports




MDW Reports allow for drill through to jump to other reports. Most reports call stored procedures in the MDW database to produce the results.

### Server Activity

This report shows the system resource consumption and amount due to SQL Server instance.

### Resource Waits

This report shows the SQL Server Activity – Batch requests/sec, compilations/sec, Transactions/sec, etc.

### Query Statistics

This report shows the expensive queries by taking delta of successive snapshots. As a result, this report does not work the same as taking the most expensive queries in sys.dm_exec_query_stats at that time.

### Completed queries during snapshot

This report shows the completed queries by CPU, Duration, Total/IO (does not include logical reads), Physical Reads, and Logical writes.

### Query Details

This report shows the resource consumption. It helps you identify query plan changes and resource consumption for each plan.

Each plan is stored in the snapshots.notable_query_plan table. If the plan changes and there is a huge difference in resource consumption, you may consider plan freezing.

### Query Plan Details

Query Execution Statistics is cumulative for all executions, so it has the same numbers as Query Details. The Query Plan Details report shows the plan size, compile time, missing indexes, parameters values, etc.

### SQL Server Waits

This report shows the drill through from Server Activity when selecting wait category. This report helps you determine which waittypes comprise a wait category. A drill through on waittype brings up different reports, such as:

- CPU – Shows queries by CPU
- Lock – Checks if blocking is occurring

Other waittypes drill to SQL Server Sampled Waits reports.

### SQL Server Sampled Waits

This report helps find the queries that were waiting on a specific waittype. These queries are filtered by wait category.

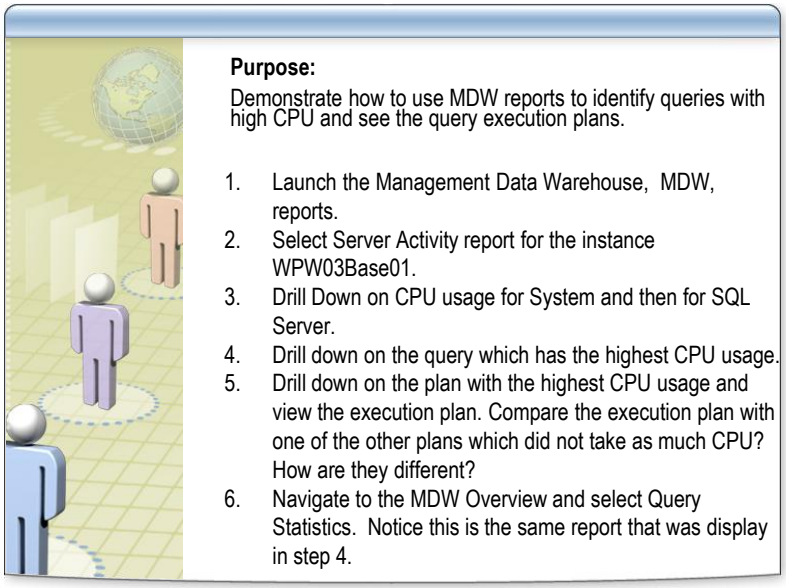
You can use this report to drill down to waittype, DB Name, Application Name, and then Query.

Note that you cannot change the hierarchy. This report is based on snapshots.rpt_sampled_waits. So, you can view the code in the procedure to generate a report with a different hierarchy.

### Disk Usage

This report shows the data and log size for each database. A drill through of the report shows data, index, unused, unallocated space, or log space (both used and unused).

# Demonstration 5: View Query with highest CPU Usage



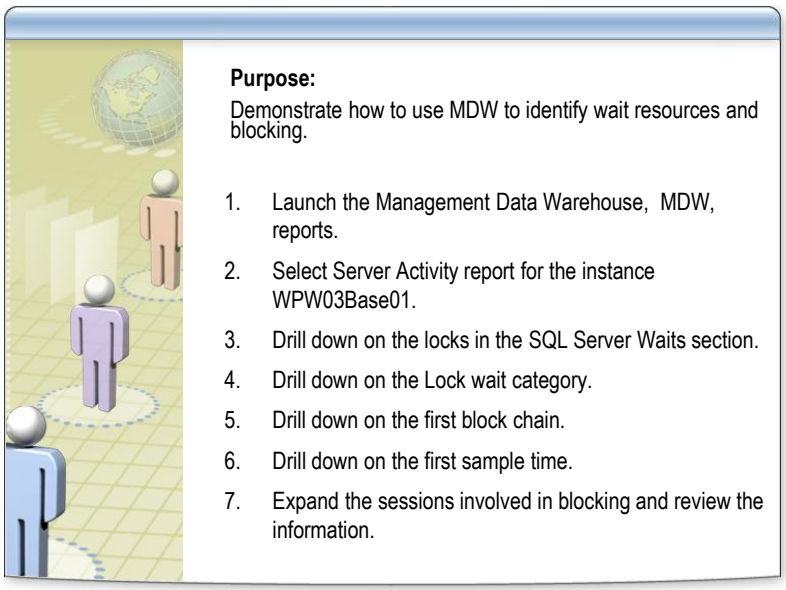
**Purpose:**

Demonstrate how to use MDW reports to identify queries with high CPU and see the query execution plans.

1. Launch the Management Data Warehouse, MDW, reports.
2. Select Server Activity report for the instance WPW03Base01.
3. Drill Down on CPU usage for System and then for SQL Server.
4. Drill down on the query which has the highest CPU usage.
5. Drill down on the plan with the highest CPU usage and view the execution plan. Compare the execution plan with one of the other plans which did not take as much CPU? How are they different?
6. Navigate to the MDW Overview and select Query Statistics. Notice this is the same report that was display in step 4.

37

# Demonstration 6: View blocking using MDW



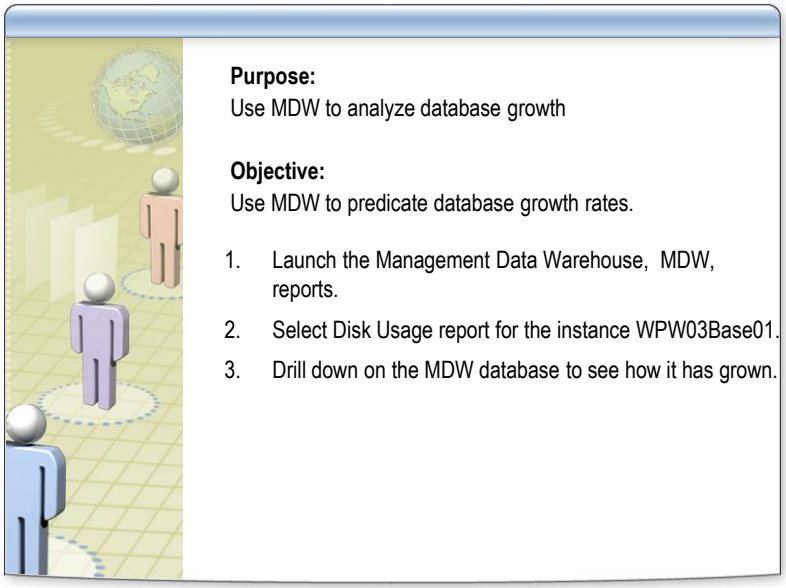
**Purpose:**

Demonstrate how to use MDW to identify wait resources and blocking.

1. Launch the Management Data Warehouse, MDW, reports.
2. Select Server Activity report for the instance WPW03Base01.
3. Drill down on the locks in the SQL Server Waits section.
4. Drill down on the Lock wait category.
5. Drill down on the first block chain.
6. Drill down on the first sample time.
7. Expand the sessions involved in blocking and review the information.

38

# Demonstration 7: View database growth using MDW

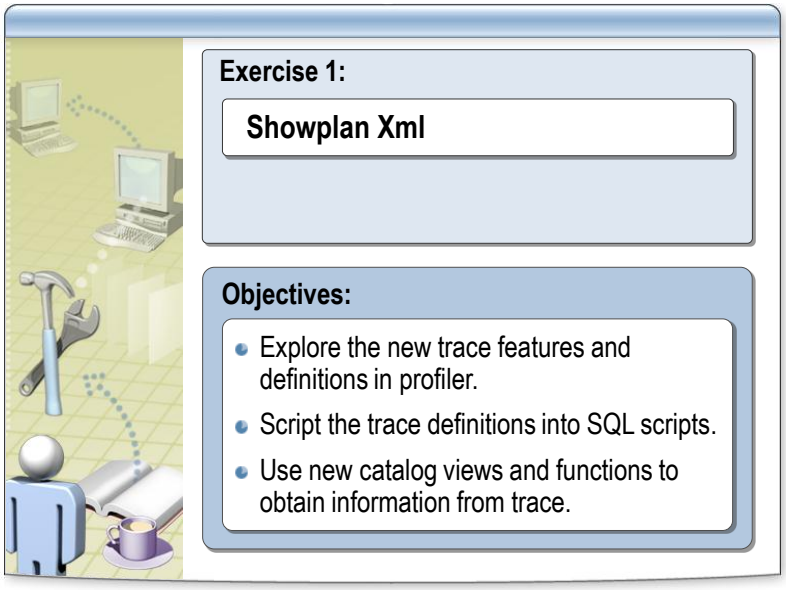


# Section 6 Review

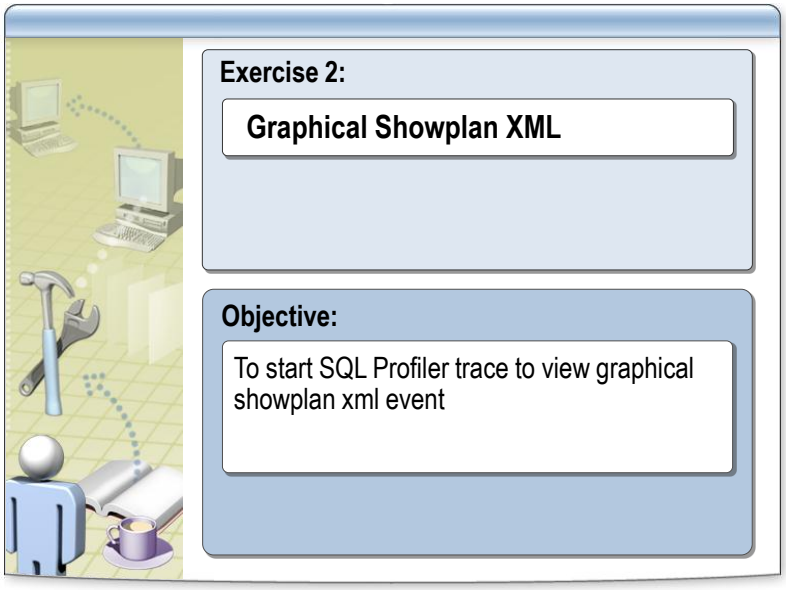
- Would you use Activity Monitor or MDW for real-time troubleshooting of SQL Server performance?
- Can Activity Monitor show if blocking is occurring, which wait category is a bottleneck, and which expensive queries have been run on the server?
- Does MDW require you to monitor the server by using other tools such as SQLDiag, PSSdiag, or SQL Profiler?

40

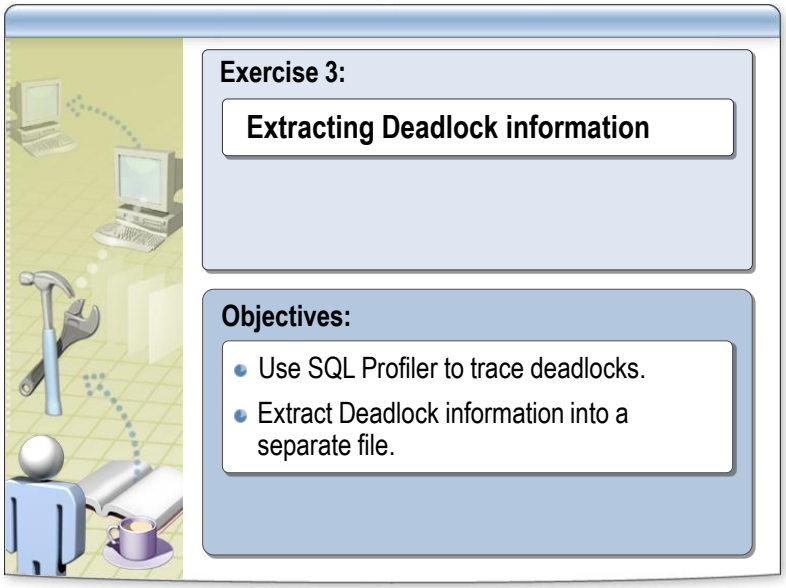
# Lab 1: Exercise 1



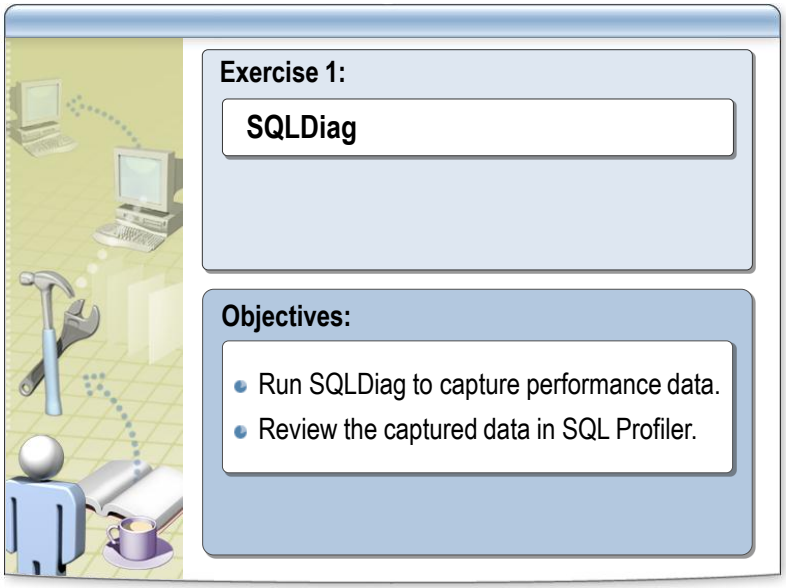
# Lab 1: Exercise 2



# Lab 1: Exercise 3

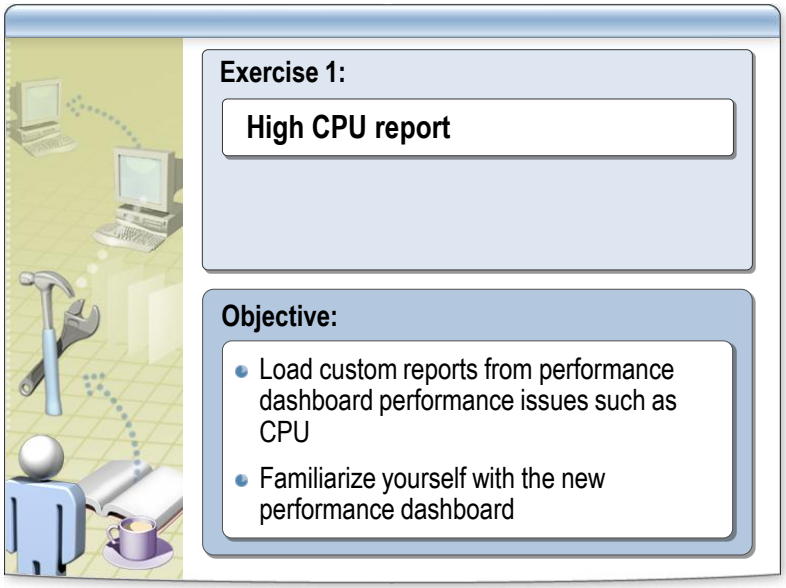


# Lab 2: Exercise 1

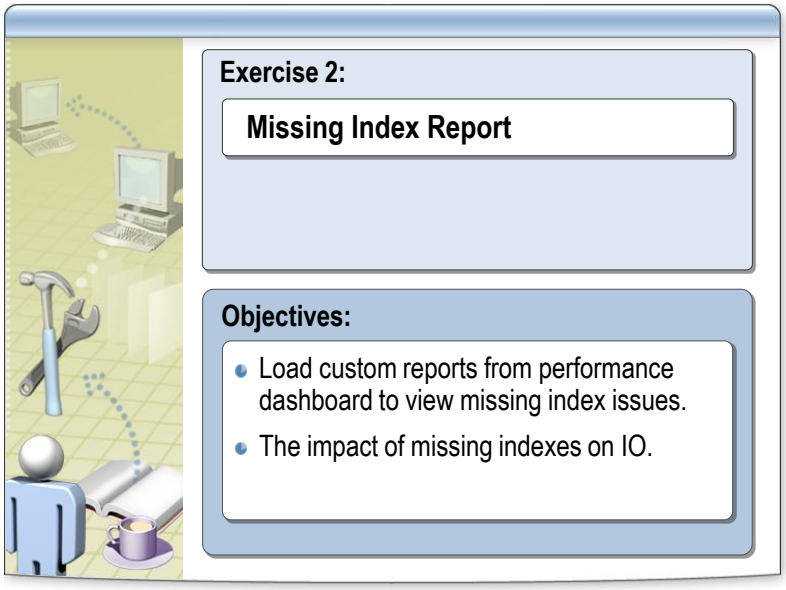



44

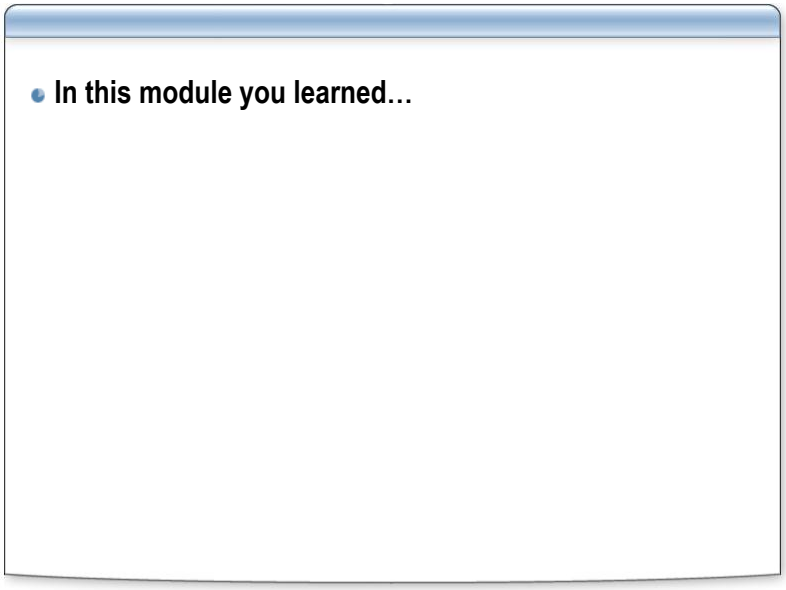
# Lab 2a: Exercise 1



# Lab 2a: Exercise 2

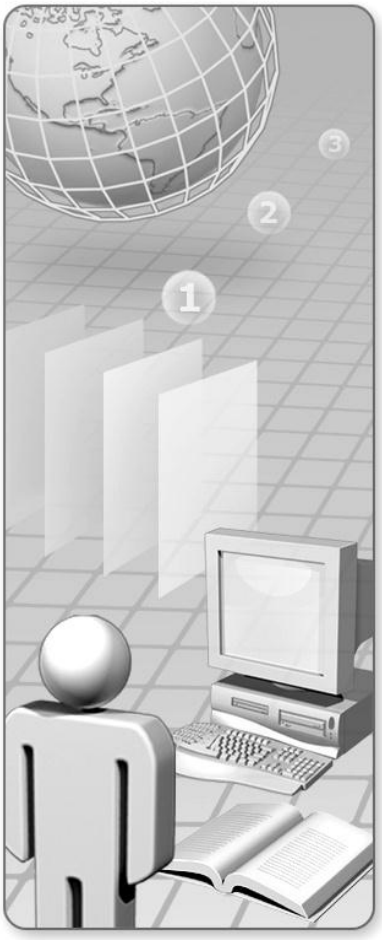


# Module Summary

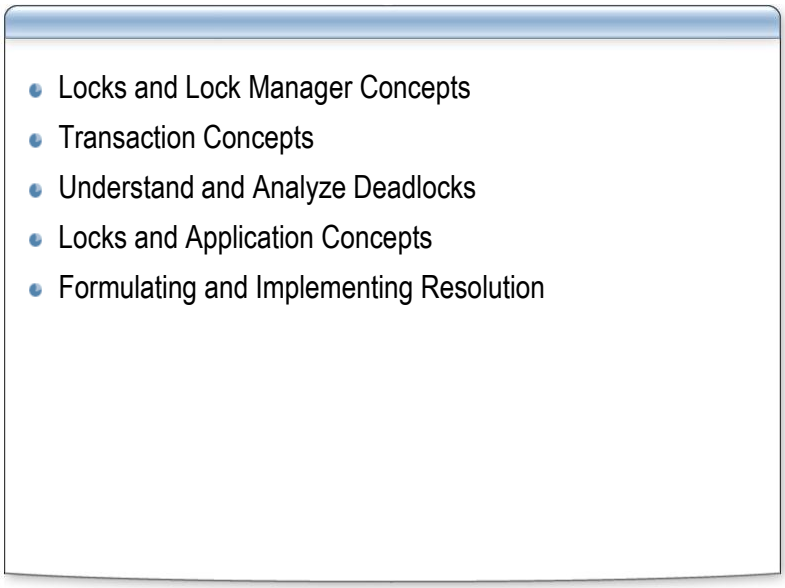



In this module, you learned about the tools that you can use to troubleshoot performance issues. You also learned which tools are used for real-time troubleshooting and which ones for monitoring SQL Server performance. In addition, you learned how to configure each tool and analyze the data collected.

# Module 4: Locking and Concurrency



# Module Overview




## Introduction

Locking is a mechanism used by the Microsoft SQL Server Database Engine to synchronize access by multiple users to the same piece of data at the same time, and prevents multiple users from changing the same data at the same time.

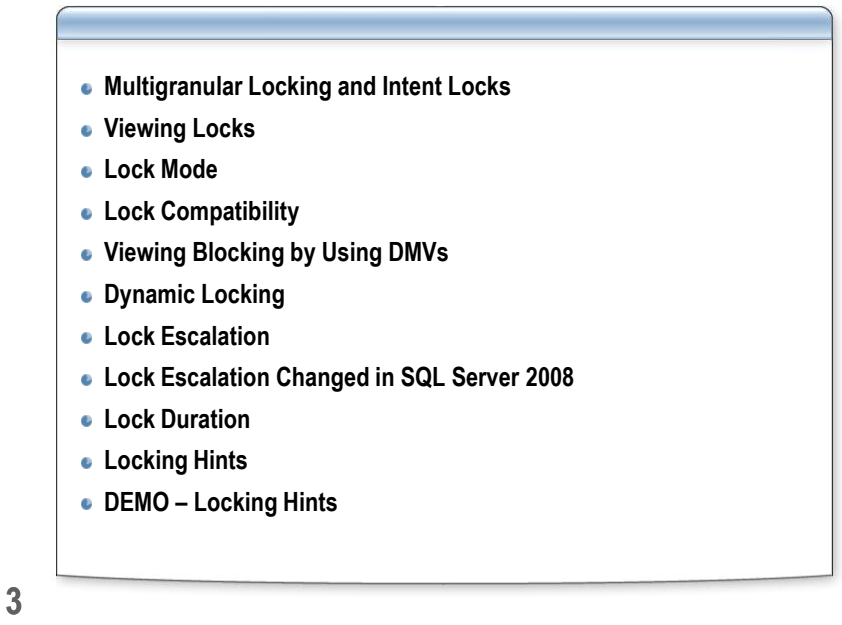
SQL Server enforces locking automatically; you can design applications that are more efficient by understanding and customizing locking in your applications.

## Objectives

After completing this module, you will be able to:

- Explain the basic concepts of locking, including lock mode, lock resources, and lock compatibility.
- Define the impact of different transaction isolation levels.
- Describe the features of transaction mode.
- Troubleshoot and resolve blocking and locking issues.

# Section 1: Locking Basics




## Introduction

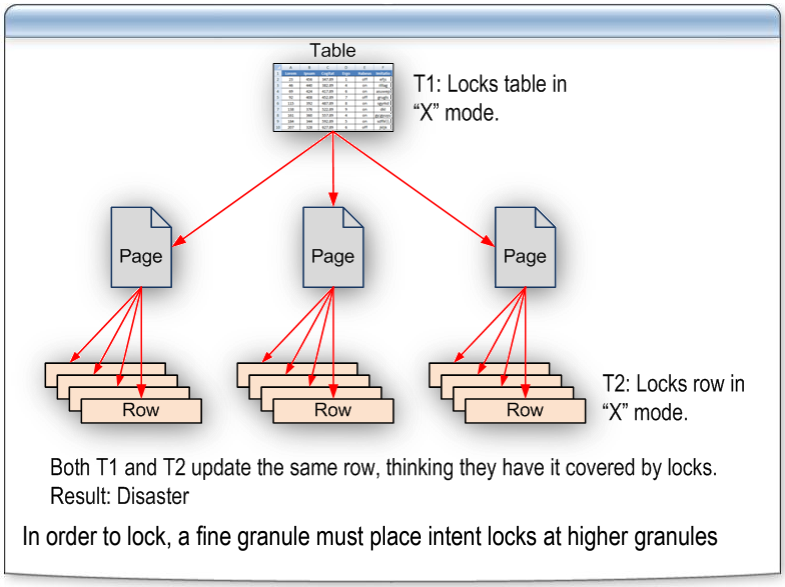
This section describes multigranular locking, various locking modes, and lock compatibility.

## Objectives

After completing this section, you will be able to:

- Explain how intent lock resolves some of the issues associated with multigranular locking.
- View lock information by using sys.dm_tran_locks.
- Define the various lock modes in SQL Server.
- Explain how lock compatibility controls whether multiple transactions can acquire locks on the same resource at the same time.
- View a blocking report by using a dynamic management view (DMV).

# Multigranular Locking and Intent Locks




## Multigranular locking

To minimize the cost of locking, SQL Server locks resources automatically at a level appropriate to the task. SQL Server has multigranular locking that allows different types of resources to be locked by a transaction.

Locking at a larger granularity, such as tables, is expensive in terms of concurrency because locking an entire table restricts access to any part of the table by other transactions, but has a lower overhead because fewer locks are being maintained.

Locking at a smaller granularity, such as rows, increases concurrency, but has a higher overhead because more locks must be held if many rows are locked.

The Database Engine often needs to acquire locks at multiple levels of granularity to fully protect a resource. This group of locks at multiple levels of granularity is called a ***lock hierarchy***.

In the example shown on the slide above, if one transaction (T1) holds an exclusive lock at the table level, and another transaction (T2) holds an exclusive lock at the row level, each of the transactions believe that they have exclusive access to the resource. In this scenario, because T1 believes that it locks the entire table, it might inadvertently make changes to the same row that T2 thought it has locked exclusively. In a multigranular locking environment, there must be a way to effectively overcome this problem. The solution is **intent lock**.

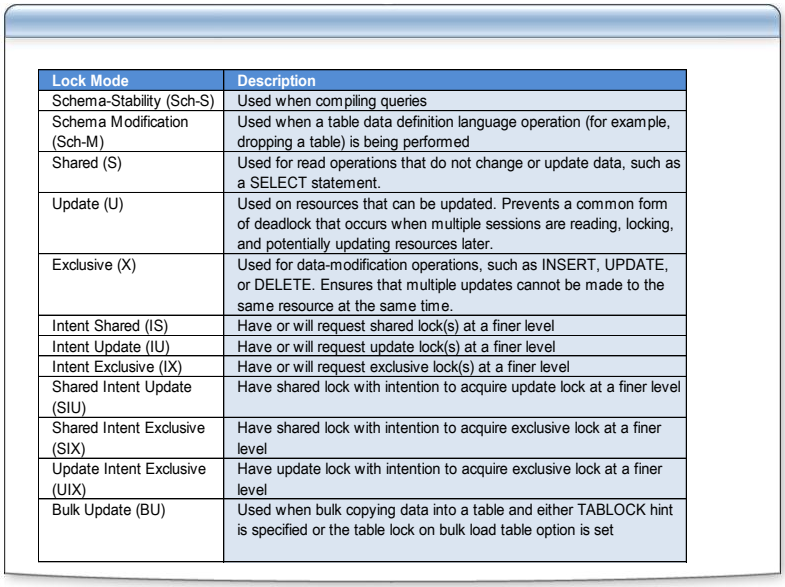
## Intent lock

Intent lock is used to establish a lock hierarchy. SQL Server Database Engine acquires low-level locks and also places intent locks on the objects that contain the lower-level objects as follows:

- When locking rows or index key ranges, intent lock is acquired on the pages that contain the rows or keys.
- When locking pages, intent lock is acquired on the higher-level objects that contain the pages. In addition to the intent lock on the object, intent page locks are requested on the following objects:
  - Leaf-level pages of non-clustered indexes
  - Data pages of clustered indexes
  - Heap data pages

An intent lock indicates that SQL Server wants to acquire a shared (S) lock or an exclusive (X) lock on some of the resources lower down in the hierarchy. For example, a shared intent lock placed at the table level means that a transaction intends on placing shared (S) locks on pages or rows within that table. Setting an intent lock at the table level prevents another transaction from subsequently acquiring an exclusive (X) lock on the table containing that page. Intent locks improve performance because SQL Server examines intent locks only at the table level to determine whether a transaction can safely acquire a lock on that table. This removes the requirement to examine every row or page lock on the table to determine whether a transaction can lock the entire table.

# Lock Mode



| Lock Mode | Description |
|---|---|
| Schema-Stability (Sch-S) | Used when compiling queries |
| Schema Modification (Sch-M) | Used when a table data definition language operation (for example, dropping a table) is being performed |
| Shared (S) | Used for read operations that do not change or update data, such as a SELECT statement. |
| Update (U) | Used on resources that can be updated. Prevents a common form of deadlock that occurs when multiple sessions are reading, locking, and potentially updating resources later. |
| Exclusive (X) | Used for data-modification operations, such as INSERT, UPDATE, or DELETE. Ensures that multiple updates cannot be made to the same resource at the same time. |
| Intent Shared (IS) | Have or will request shared lock(s) at a finer level |
| Intent Update (IU) | Have or will request update lock(s) at a finer level |
| Intent Exclusive (IX) | Have or will request exclusive lock(s) at a finer level |
| Shared Intent Update (SIU) | Have shared lock with intention to acquire update lock at a finer level |
| Shared Intent Exclusive (SIX) | Have shared lock with intention to acquire exclusive lock at a finer level |
| Update Intent Exclusive (UIX) | Have update lock with intention to acquire exclusive lock at a finer level |
| Bulk Update (BU) | Used when bulk copying data into a table and either TABLOCK hint is specified or the table lock on bulk load table option is set |

5

SQL Server Database Engine locks resources by using different lock modes that determine how the resources can be accessed by concurrent transactions.

SQL Server sets the lock mode on the following resources:

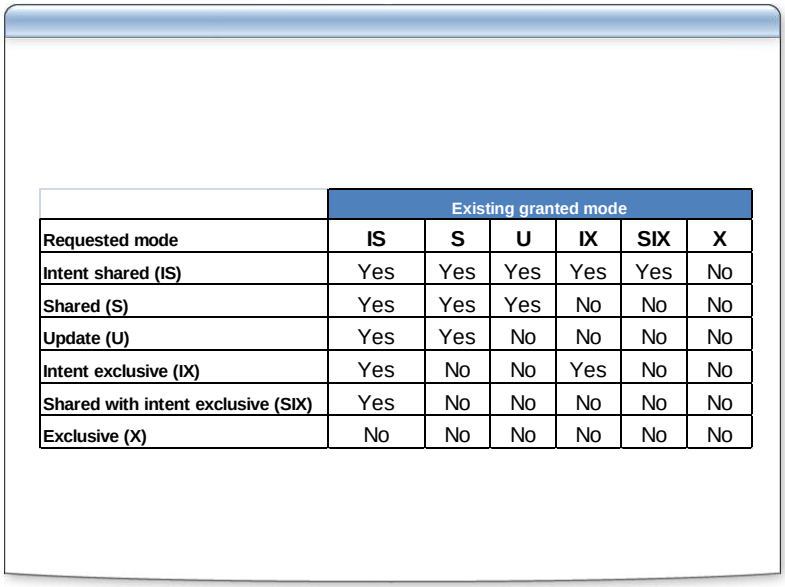
- A database
- A database file
- An entire table, including all data and indexes
- An 8-kilobyte (KB) data page or index page
- Row lock within an index
- Metadata information
- Internal storage structures, such as heap or B-tree (HoBT) structures
- Allocation units
- A lock resource defined by an application

## Lock modes

- **Schema modification (Sch-M) locks** are taken during a table data definition language (DDL) operation. A Sch-M lock prevents concurrent access to the table. This lock blocks all outside operations until the lock is released. You should use Sch-M locks to prevent access to affected tables by concurrent operations.

- **Schema stability (Sch-S) locks** are taken when compiling and running queries. Sch-S locks do not block any transactional locks, including exclusive (X) locks. Transactions continue to run while a query is being compiled.
- **Shared (S) locks** allow concurrent transactions to read (SELECT) a resource under pessimistic concurrency control. However, no other transactions can modify the data while shared (S) locks exist on the resource.
- **Update (U) locks** prevent a common form of deadlock. Only one transaction can obtain an update (U) lock to a resource at a time. If a transaction modifies a resource, the update (U) lock is converted to an exclusive (X) lock.
- **Exclusive (X) locks** prevent access to a resource by concurrent transactions. With an exclusive (X) lock, by default, no other transactions can modify data.
- **Intent shared (IS) lock** protects requested or acquired shared locks on some resources lower in the hierarchy.
- **Intent exclusive (IX) lock** protects requested or acquired exclusive locks on some resources lower in the hierarchy. IX is a superset of IS, and it also protects requesting shared locks on lower-level resources.
- **Shared with intent exclusive (SIX) lock** protects requested or acquired shared locks on all resources lower in the hierarchy and intent exclusive locks on some of the lower level resources. Concurrent IS locks at the top-level resource are allowed.
- **Intent update (IU) lock** protects requested or acquired update locks on all resources lower in the hierarchy. IU locks are used only on page resources. If the event of an update operation, IU locks are converted to IX locks.
- **Shared intent update (SIU) lock** is a combination of S and IU locks, as a result of acquiring these locks separately and simultaneously holding both locks.
- **Update intent exclusive (UIX) lock** is a combination of U and IX locks, as a result of acquiring these locks separately and simultaneously holding both locks.
- **Bulk update (BU) locks** allow multiple threads to bulk load data concurrently into the same table while preventing other processes that are not bulk loading data from accessing the table. For example, using bulk update (BU) locks when bulk copying data into a table.

# Lock Compatibility



| | Existing granted mode | | | | | |
|---|---|---|---|---|---|---|
| **Requested mode** | **IS** | **S** | **U** | **IX** | **SIX** | **X** |
| **Intent shared (IS)** | Yes | Yes | Yes | Yes | Yes | No |
| **Shared (S)** | Yes | Yes | Yes | No | No | No |
| **Update (U)** | Yes | Yes | No | No | No | No |
| **Intent exclusive (IX)** | Yes | No | No | Yes | No | No |
| **Shared with intent exclusive (SIX)** | Yes | No | No | No | No | No |
| **Exclusive (X)** | No | No | No | No | No | No |

6

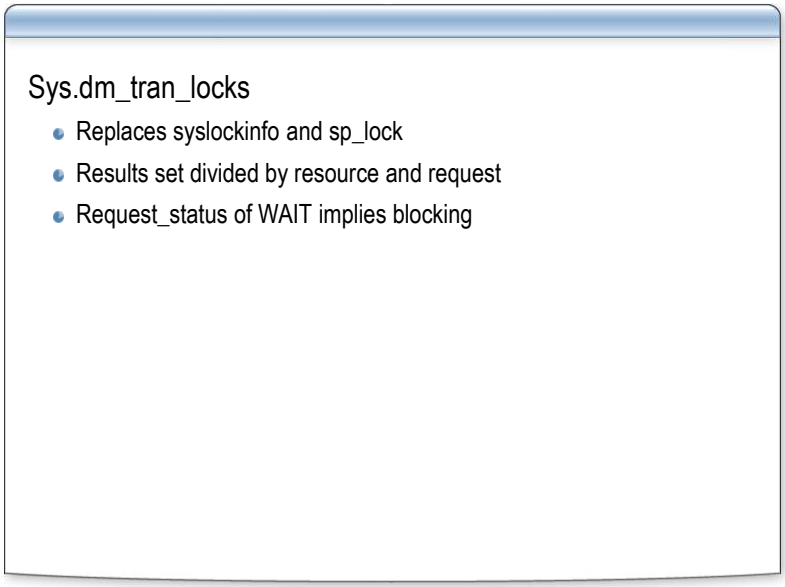
An intent exclusive (IX) lock is compatible with an IX lock mode because IX means that the intention is to update only some of the rows, rather than all of them. Other transactions that attempt to read or update some of the rows are also permitted as long as they are not the same rows being updated by other transactions.

IX locks can be applied at any level of granularity above the leaf level. If a row is locked, SQL Server will apply intent locks at both the page and the table level. If a page is locked, SQL Server will apply an intent lock at the table level.

SIX locks imply that there is a shared access to a resource and tthat here are X locks placed at a lower level in the hierarchy. SQL Server never asks for SIX locks directly; they are always the result of a conversion. For example, suppose a transaction scanned a page by using an S lock and then subsequently decided to perform a row-level update. The row would obtain an X lock, but now the page would require an IX lock. The resultant mode on the page would be SIX.

> **Note**: For a complete matrix of lock compatibility, refer to *Lock Compatibility (Database Engine)* in SQL Server Books Online.

# Viewing Locks



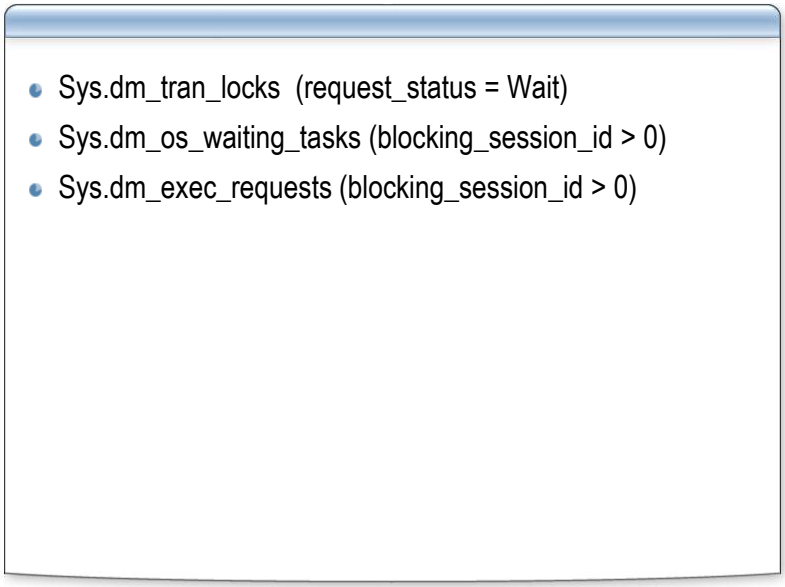

In SQL Server 2005 and SQL Server 2008, the **sys.dm_tran_locks** dynamic management view (DMV) replaces **syslockinfo** and **sp_lock** from SQL 2000. This DMV returns information about currently active lock manager resources, including both the resources and the requests. Each row represents a currently active request to the lock manager for a lock that has been granted or is waiting to be granted.

The **sys.dm_tran_locks** DMV is populated from internal lock manager data structures; maintaining this information does not add extra overhead to regular processing. However, materializing the view does require access to the lock manager internal data structures. This can have minor effects on the regular processing in the server. These effects should be unnoticeable and should only affect heavily used resources. The data in this view corresponds to live lock manager state, the data can change at any time, and rows are added and removed as locks are acquired and released. This view does not have any historical information.

A GRANTED request status indicates that a lock has been granted on a resource to the requestor. A WAITING request status indicates that the request has not yet been granted. The following waiting-request types are returned by the **request_status** column:

- CONVERT. This request status indicates that the requestor has already been granted a request for the resource and is currently waiting for an upgrade to the initial request to be granted.
- WAIT. This request status indicates that the requestor does not currently hold a granted request on the resource.

# Viewing Blocking by Using DMVs




You can use the following DMVs to view information regarding blocked resources and the cause for blocking:

- **sys.dm_tran_locks**: While using this DMV, check the *request_status* column value for the term **Wait**. This indicates that the requested lock mode has not been granted and is waiting.
- **sys.dm_os_waiting_tasks**: While using this DMV, check the *blocking_session_id*. A value greater than 0 indicates that the session_id is blocking the task.
- **sys.dm_exec_request**: While using this DMV, check the *blocking_session_id*. A value greater than 0 indicates that the session_id is blocking the session.

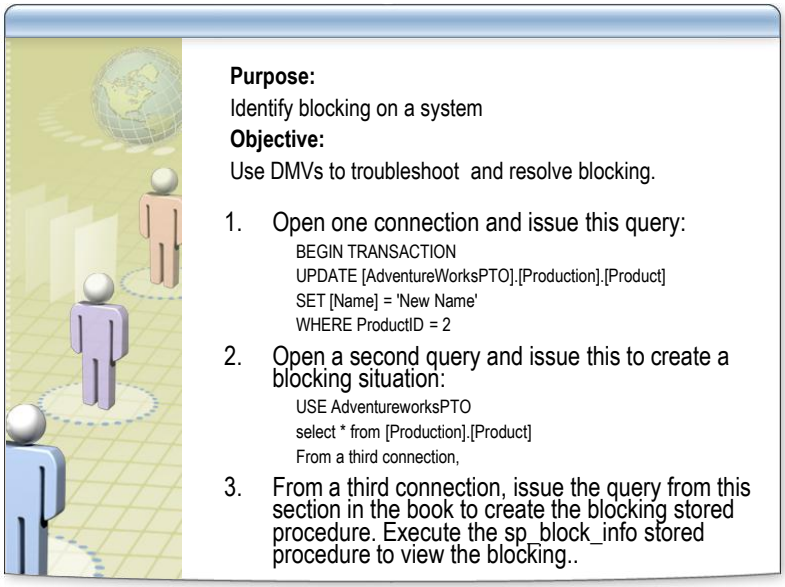
The following code generates a blocking report by using DMVs:

```
create proc sp_block_info as
select t1.resource_type as [lock type] ,db_name(resource_database_id) as
[database]
,t1.resource_associated_entity_id as [blk object]
,t1.request_mode as [lock req]                        -- lock requested
,t1.request_session_id as [waiter sid]                -- spid of waiter
,t2.wait_duration_ms as [wait time]
,(select text from sys.dm_exec_requests as r          --- get sql for waiter
cross apply sys.dm_exec_sql_text(r.sql_handle)
where r.session_id = t1.request_session_id) as waiter_batch
,(select substring(qt.text,r.statement_start_offset/2,
(case when r.statement_end_offset = -1 then len(convert(nvarchar(max),
               qt.text)) * 2
else r.statement_end_offset end - r.statement_start_offset)/2)
from sys.dm_exec_requests as r
```

```
cross apply sys.dm_exec_sql_text(r.sql_handle) as qt
where r.session_id = t1.request_session_id) as waiter_stmt    --- statement
executing now
,t2.blocking_session_id as [blocker sid]            --- spid of blocker
,(select text from sys.sysprocesses as p            --- get sql for blocker
cross apply sys.dm_exec_sql_text(p.sql_handle)
where p.spid = t2.blocking_session_id) as blocker_stmt
from sys.dm_tran_locks as t1, sys.dm_os_waiting_tasks as t2
where t1.lock_owner_address = t2.resource_address
```

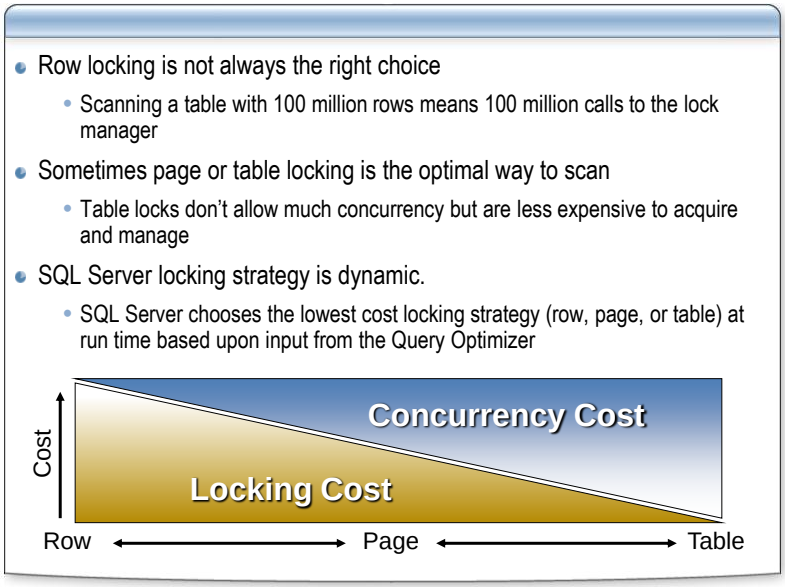
**Note**: You can find the procedure shown above at http://blogs.msdn.com/sqlcat/archive/2005/12/12/502735.aspx as well.

# Demonstration 1: Blocking DMVs



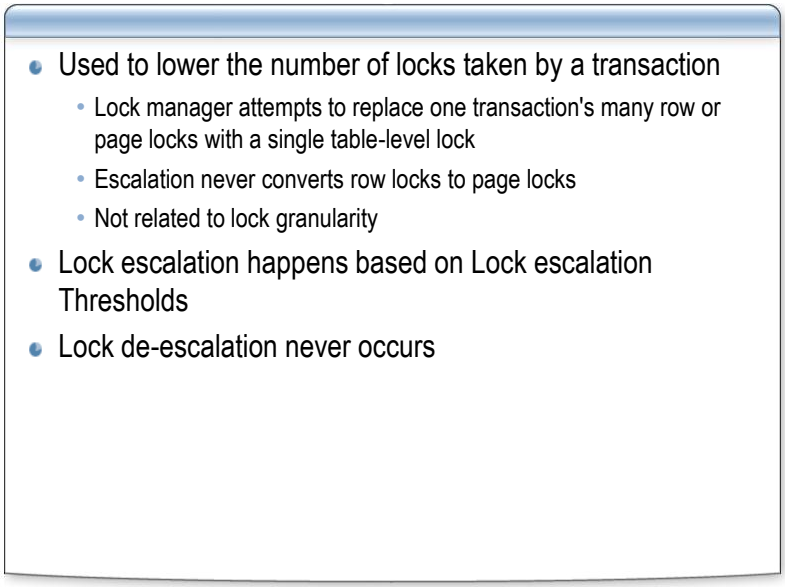

9

# Dynamic Locking




When modifying individual rows, SQL Server typically takes row locks to maximize concurrency (for example, OLTP and order-entry applications). When scanning larger volumes of data, it would be more appropriate to take page or table locks to minimize the cost of acquiring locks (for example, Decision Support System (DSS), data warehouse, and reporting).

# Lock Escalation




When the lock count for a transaction is a multiple of the escalation threshold, the lock manager attempts to escalate the locks. The number of locks held may continue to increase after the escalation attempt (for example, because new tables are accessed, or the previous lock escalation attempts failed due to incompatible locks held by another SPID (server process ID)).

The lock manager checks the lock memory that it is using, and if it is more than 40 percent of the allocated buffer pool memory of SQL Server, it tries to find a scan where no escalation has already been performed. The lock manager repeats this search operation until all scans have been escalated or until the memory consumption drops.

## Lock escalation thresholds

Lock escalation can be triggered in any of the following situations:

- When a single Transact-SQL (T-SQL) statement acquires at least 5,000 locks on a single table or index.
- When the number of locks in an instance of the Database Engine exceeds memory or configuration thresholds.
- If locks cannot be escalated because of lock conflicts, the Database Engine periodically triggers lock escalation at every 1,250 new locks acquired.

### Escalation threshold for a T-SQL statement

- Lock escalation is triggered when a T-SQL statement acquires at least 5,000 locks on a single reference of a table or index, or, if the table is partitioned, a single reference of a table partition or index partition. For example, lock escalation is not triggered if

a statement acquires 3,000 locks in one index and 3,000 locks in another index of the same table. Similarly, lock escalation is not triggered if a statement has a self join on a table, and each reference to the table only acquires 3,000 locks in the table.

- Lock escalation only occurs for tables that have been accessed at the time that the escalation is triggered. Assume that a single SELECT statement is a join that accesses three tables in the following sequence: TableA, TableB, and TableC. The statement acquires 3,000 row locks in the clustered index for TableA and at least 5,000 row locks in the clustered index for TableB, but has not yet accessed TableC. When the Database Engine detects that the statement has acquired at least 5,000 row locks in TableB, it attempts to escalate all locks held by the current transaction on TableB. It also attempts to escalate all locks held by the current transaction on TableA, but because the number of locks on TableA is less than 5000, the escalation will not succeed. Lock escalation for TableC is not attempted because it had not yet been accessed when the escalation occurred.

### Escalation threshold for an instance of the database engine

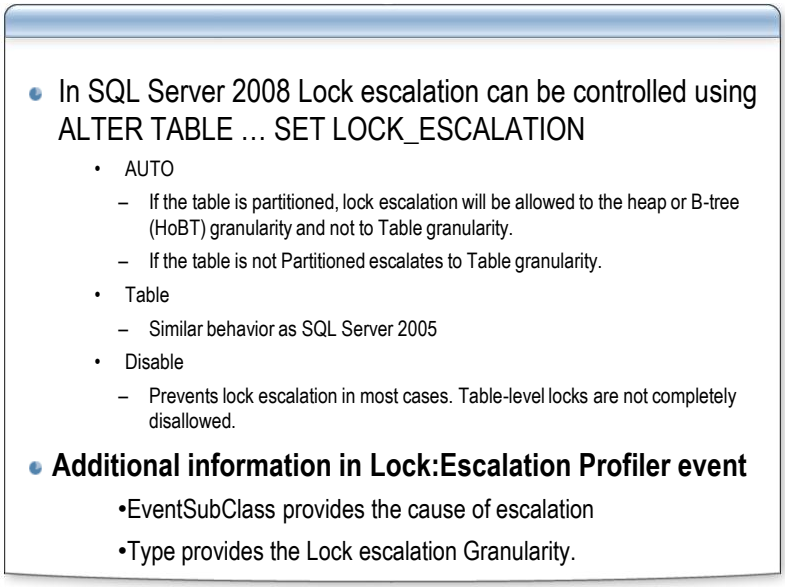
Whenever the number of locks is greater than the memory threshold for lock escalation, the Database Engine triggers lock escalation. The memory threshold depends on the following setting of the locks configuration option:

- If the locks option is set to its default setting of 0, then the lock escalation threshold is reached when the memory used by lock objects is 24 percent of the memory used by the Database Engine, excluding Address Windowing Extensions (AWE) memory. The data structure used to represent a lock is approximately 100 bytes long. This threshold is dynamic because the Database Engine dynamically acquires and frees memory to adjust for varying workloads.
- If the locks option is a value other than 0, then the lock escalation threshold is 40 percent (or less if there is a memory pressure) of the value of the locks option.

The Database Engine can choose any active statement from any session for escalation, and for every 1,250 new locks, it will choose statements for escalation as long as the lock memory used in the instance remains above the threshold.

Customers can continue to influence lock escalation behavior via the trace flags -T1211. This disables the lock escalation for the instance. Enabling -T1211 does not prevent SQL Server from choosing page or table lock rather than row lock.

# Lock Escalation Changes in SQL Server 2008




In SQL Server 2008, lock escalation can be controlled using ALTER TABLE (T-SQL Syntax).

In the ALTER TABLE (T-SQL Syntax), lock escalation can be set to specify the allowed methods of lock escalation for a table as follows:

```
SET ( LOCK_ESCALATION = { AUTO | TABLE | DISABLE } )
```

In the above syntax:

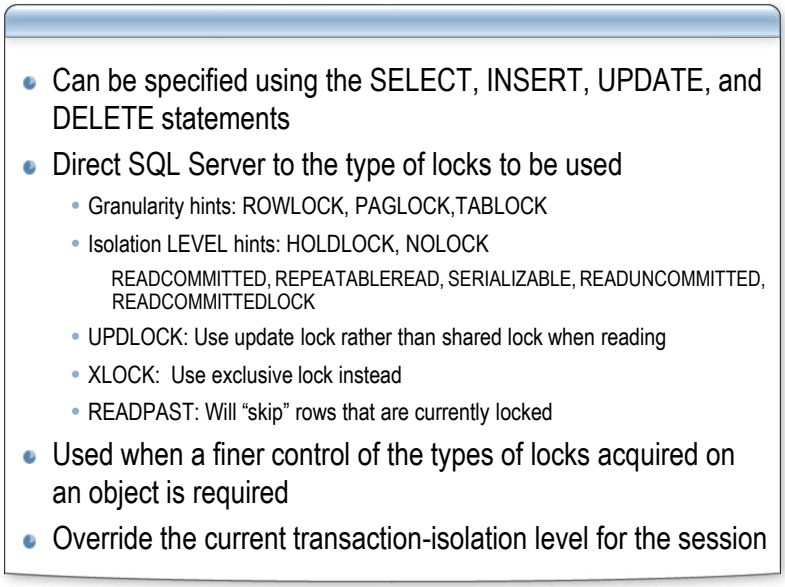
- AUTO: This option allows the SQL Server Database Engine to select the lock escalation granularity that is appropriate for the table schema. If the table is partitioned, lock escalation will be allowed to the heap or B-tree (HoBT) granularity. After the lock is escalated to the HoBT level, the lock will not be escalated later to the TABLE granularity. If the table is not partitioned, the lock escalation will be done to the TABLE granularity.
- TABLE: This option specifies that lock escalation will be done at table-level granularity regardless of whether the table is partitioned or not partitioned. This behavior is the same as in SQL Server 2005; TABLE is the default value.
- DISABLE: This option prevents lock escalation in most cases. Table-level locks are not completely disallowed. For example, when you are scanning a table that does not have any clustered index under the serializable isolation level, the Database Engine must take a table lock to protect data integrity.

## Lock: Escalation event class

In SQL Server 2008, the Lock: Escalation event class has been enhanced to provide the following additional information:

- **EventSubClass**: Provides the cause of the lock escalation as follows:
  - 0 - LOCK_THRESHOLD indicates that the statement exceeded the lock threshold.
  - 1 - MEMORY_THRESHOLD indicates that the statement exceeded the memory threshold.
- **Type**: Provides the lock escalation granularity as follows:
  - 1=NULL_RESOURCE
  - 2=DATABASE
  - 3=FILE
  - 5=OBJECT (table level)
  - 6=PAGE
  - 7=KEY
  - 8=EXTENT
  - 9=RID
  - 10=APPLICATION
  - 11=METADATA
  - 12=HOBT
  - 13=ALLOCATION_UNIT

# Locking Hints




Locking hints specify the type of locking or row versioning that the instance of the Microsoft SQL Server Database Engine uses for the table data. These locking hints override the current transaction isolation level for the session.

You can specify locking hints for individual table references in the SELECT, INSERT, UPDATE, and DELETE statements.

### UPDLOCK

If update locks are used instead of shared locks while reading a table, the locks are held until the end of the statement or transaction. UPDLOCK has the advantage of enabling you to read data (without blocking other readers) and update it later with the assurance that the data has not changed since you last read it.

### READCOMMITTED

READCOMMITTED has been modified to account for READ COMMITTED SNAPSHOT isolation. If the database option READ_COMMITTED_SNAPSHOT is OFF, the Database Engine acquires shared locks as data is read and releases these locks when the read operation is completed. If the database option READ_COMMITTED_SNAPSHOT is ON, the Database Engine does not acquire locks and uses row versioning instead.

### READCOMMITTEDLOCK

The Database Engine acquires shared locks as data is read and releases these locks when the read operation is completed, regardless of the setting of the READ_COMMITTED_SNAPSHOT database option.

### READPAST

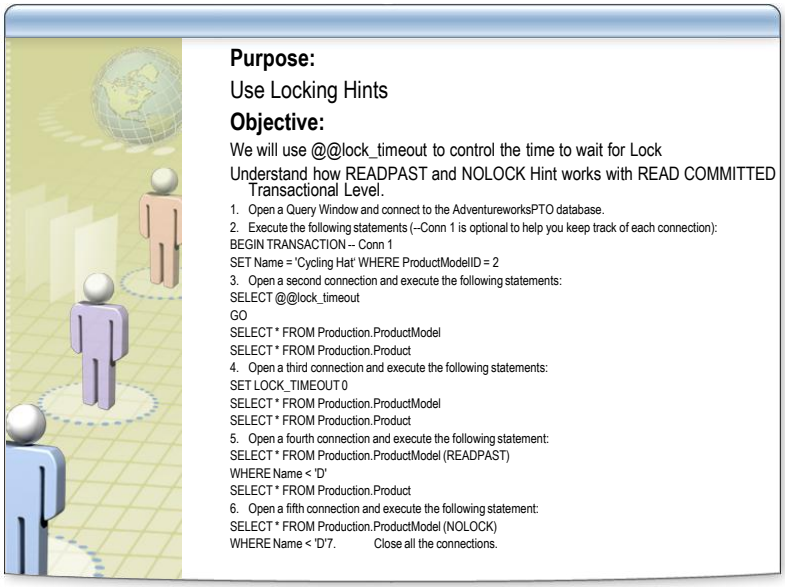
READPAST is an optimizer hint for use with SELECT statements. When this hint is used, SQL Server will read past locked rows. For example, assume that table T1 contains a single integer column with the values of 1, 2, 3, 4, and 5. If transaction A changes the value of 3 to 8 but has not yet committed, a SELECT * FROM T1 (READPAST) yields the values 1, 2, 4, 5.

> **Tip:** With rare exception, READPAST only applies to transactions operating at READ COMMITTED or REPEATABLE READ isolation levels and only reads past row-level locks.

You can use the READPAST lock hint to implement a work queue on a SQL Server table. For example, assume that there are a bunch of external work requests being thrown into a table and that they should be serviced in approximate insertion order, but they do not have to be completely first-in, first-out (FIFO). If you have four worker threads consuming work items from the queue, they could each pick up a record by using READPAST locking and then delete the entry from the queue and commit when they are done. If they fail, they could roll back, leaving the entry on the queue for the next worker thread to pick up.

> **Caution:** The READPAST hint is not compatible with HOLDLOCK.

# Demonstration 2: Locking Hints (Optional)




You can use locking hints when you require a finer control of the types of locks acquired on an object.

## Try This: Using locking hints

In this demonstration, you will use @@lock_timeout to control the time to wait for lock. You will also understand how READPAST and NOLOCK hints work with the READ COMMITTED transactional level.

1. Open a **Query** window and connect to the **pubs** database.
2. Run the following statements (--Conn 1 is optional to help you keep track of each connection):

```
BEGIN TRANSACTION -- Conn 1
UPDATE titles
SET price = price * 0.9
WHERE title_id = 'BU1032'
```

3. Open a second connection, and run the following statements:

```
SELECT @@lock_timeout -- Conn 2
       GO
SELECT * FROM titles
SELECT * FROM authors
```

4. Open a third connection, and run the following statements:

```
SET LOCK_TIMEOUT 0 -- Conn 3
SELECT * FROM titles
SELECT * FROM authors
```

5. Open a fourth connection, and run the following statements:

```
SELECT * FROM titles (READPAST) -- Conn 4
WHERE title_ID < 'C'
SELECT * FROM authors
```

   How many records were returned?

6. Open a fifth connection, and run the following statements:

```
SELECT * FROM titles (NOLOCK) -- Conn 5
WHERE title_ID < 'C'
SELECT * FROM authors
```

   How many records were returned?

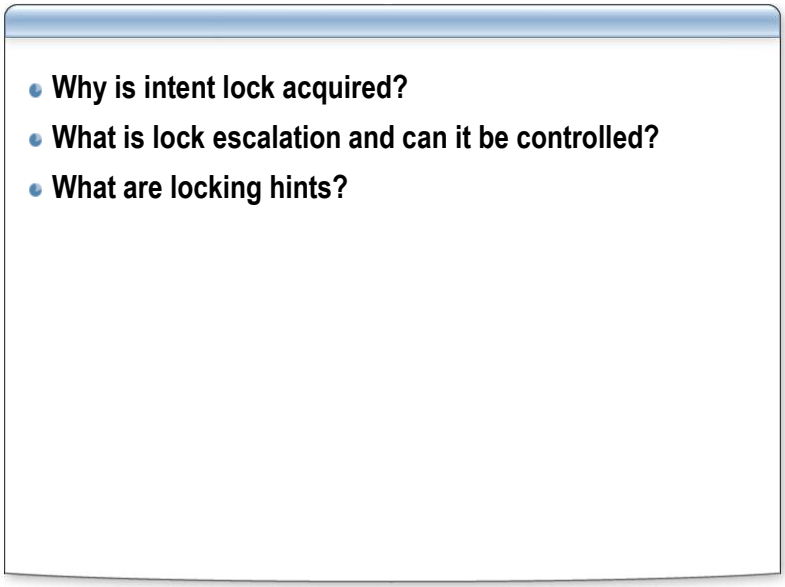
7. Close all the connections.

## Questions

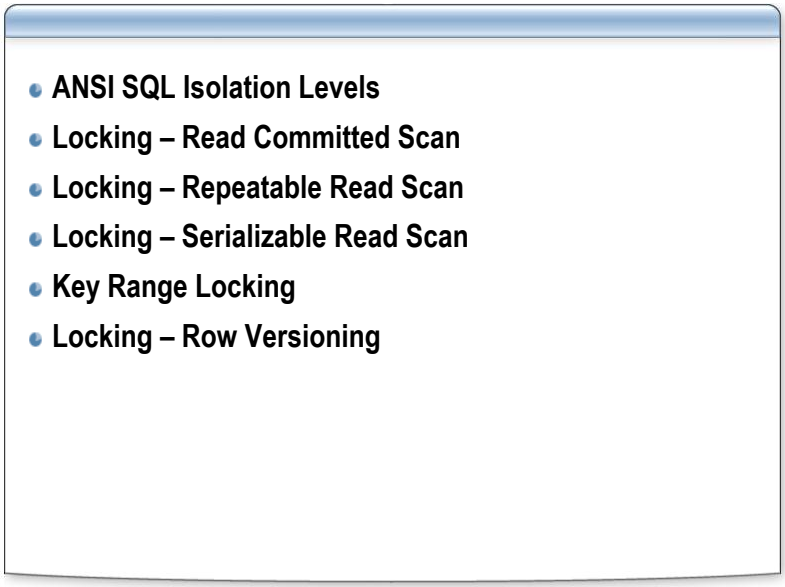
- What happened when lock timeout was left at the default?
- What was the difference between setting the timeout to 0 and using the READPAST hint?
- How was READPAST different than NOLOCK?

> **Note:** The Database Engine query optimizer almost always chooses the correct locking level. We recommend using table-level locking hints to change the default locking behavior only when necessary. Disallowing a locking level can adversely affect concurrency.

# Section 1 Review



# Section 2: Isolation Levels and Locking




## Introduction

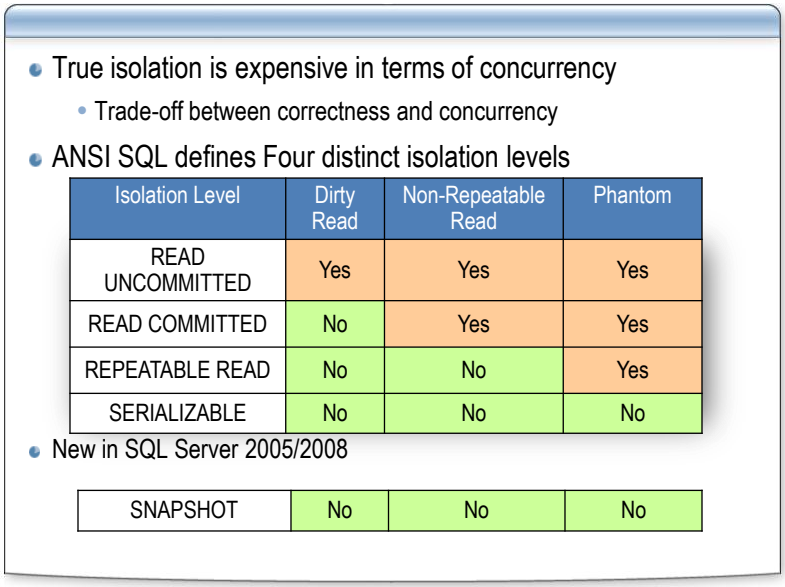
This section explains the isolation level and row versioning-based isolation level concepts.

## Objectives

After completing this section, you will be able to:

- Set and verify the isolation level of a session.
- Explain how SQL Server handles shared locks in a read committed, repeatable read, and serializable scan.
- Enable row versioning-based isolation levels and manage information update.

# ANSI SQL Isolation Levels




An isolation level that defines the degree to which one transaction must be isolated from resource or data modifications made by other transactions. Isolation levels are described in terms of which concurrency side-effects, such as dirty reads or phantom reads, are allowed.

By default, SQL Server operates at an isolation level of READ COMMITTED. To make use of either more or less-strict isolation levels in applications, you can customize locking for an entire session by setting the isolation level of the session with the SET TRANSACTION ISOLATION LEVEL statement.

To determine the transaction isolation level currently set, you can use the DBCC USEROPTIONS statement as follows:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
GO
DBCC USEROPTIONS
GO
```

SQL Server 2005 introduces a new DMV that you can use to verify the current isolation level. Run the following statement to query **sys.dm_exec_sessions** and check the column **transaction_isolation_level** for each opened session:

```
SELECT session_id, transaction_isolation_level FROM sys.dm_exec_sessions
```

The query above returns a value representing the transaction isolation level of the session, where:

- 0 = Unspecified
- 1 = READ UNCOMMITTED
- 2 = READ COMMITTED (default)
- 3 = REPEATABLE
- 4 = SERIALIZABLE
- 5 = SNAPSHOT

The following sections describe each of these transaction isolation levels listed above:

### READ UNCOMMITTED

Statements can read rows that have been modified by other transactions but not yet committed. This will not prevent *dirty reads*.

Transactions running at the READ UNCOMMITTED level do not issue shared locks to prevent other transactions from modifying the data read by the current transaction. READ UNCOMMITTED transactions are also not blocked by exclusive locks that would prevent the current transaction from reading rows that have been modified but not committed by other transactions.

### READ COMMITTED

Data can be changed by other transactions between individual statements within the current transaction, resulting in nonrepeatable reads or phantom data. Statements cannot read data that has been modified but not committed by other transactions. This prevents dirty reads.

### REPEATABLE READ

Statements cannot read data that has been modified but not yet committed by other transactions. In addition, no other transactions can modify the data that has been read by the current transaction until the current transaction completes.

### SERIALIZABLE

Statements cannot read the data that has been modified but not yet committed by other transactions.

No other transactions can modify the data that has been read by the current transaction until the current transaction completes.

Other transactions cannot insert new rows with key values that would fall in the range of keys read by any statements in the current transaction until the current transaction completes.

### SNAPSHOT

Snapshot is a new transaction isolation level in SQL Server 2005 and SQL Server 2008. It specifies that the data read by any statement in a transaction will be the transitionally consistent version of the data that existed at the start of the transaction. The transaction can only recognize data modifications that were committed before the start of the

transaction. Data modifications made by other transactions after the start of the current transaction are not visible to statements running in the current transaction. The effect is as if the statements in a transaction get a snapshot of the committed data as it existed at the start of the transaction.

The ALLOW_SNAPSHOT_ISOLATION database option must be set to ON before you can start a transaction that uses the SNAPSHOT isolation level. If a transaction using the SNAPSHOT isolation level accesses data in multiple databases, ALLOW_SNAPSHOT_ISOLATION must be set to ON in each database.

The difference between SNAPSHOT and SERIALIZABLE is that SERIALIZABLE holds locks to prevent dirty, non-repeatable, and phantom reads. The SNAPSHOT isolation level does not hold locks.

To enable the snapshot isolation level:

1. Enable the database for snapshot isolation as follows:

```
ALTER DATABASE DBNAME
SET ALLOW_SNAPSHOT_ISOLATION ON
```

2. Set the transaction isolation level in the session as follows:

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
```

If a session sets SNAPSHOT isolation ON, but the database option is not set, the session receives an error when you issue a SELECT statement. The error returned is:

```
Msg 3952, Level 16, State 1, Line 1
Snapshot isolation transaction failed accessing database 'AdventureWorks_PTO'
because snapshot isolation is not allowed in this database. Use ALTER DATABASE to
allow snapshot isolation.
```

When you set ALLOW_SNAPSHOT_ISOLATION to a new state (from ON to OFF, or from OFF to ON), ALTER DATABASE does not return control to the caller until all existing transactions in the database are committed.

You can use the following values in the sys.databases column snapshot_isolation_state_desc to determine the current state of snapshot isolation:

| Value | Description |
|---|---|
| 0 | Snapshot isolation state is OFF (default). Snapshot isolation is disallowed. |
| 1 | Snapshot isolation state is ON. Snapshot isolation is allowed. |
| 2 | Snapshot isolation state is in transition to OFF state. All transactions have their modifications versioned. New transactions cannot be started using snapshot isolation. The database remains in the transition to OFF state until all transactions that were active when ALTER DATABASE was run can be completed. |
| 3 | Snapshot isolation state is in transition to ON state. New transactions have their modifications versioned. Transactions cannot use snapshot isolation until the snapshot isolation state becomes 1 (ON). The database remains in the transition to ON state until all update transactions that were active when ALTER DATABASE was run can be |

| Value | Description |
|---|---|
| | completed. |

## READ_COMMITTED_SNAPSHOT

The Read Committed isolation level, when used with the READ_COMMITTED_SNAPSHOT database option set to ON, operates the same as snapshot with regard to dirty reads. The difference is that when using these options, the SELECT statement will not be blocked due to a write and the last committed value for the row is returned. Typically, if you use the READ COMMITTED isolation level and the row is locked, blocking occurs to prevent a dirty read.

To enable the Read Committed isolation level:

1. Enable the database for snapshot isolation as follows:

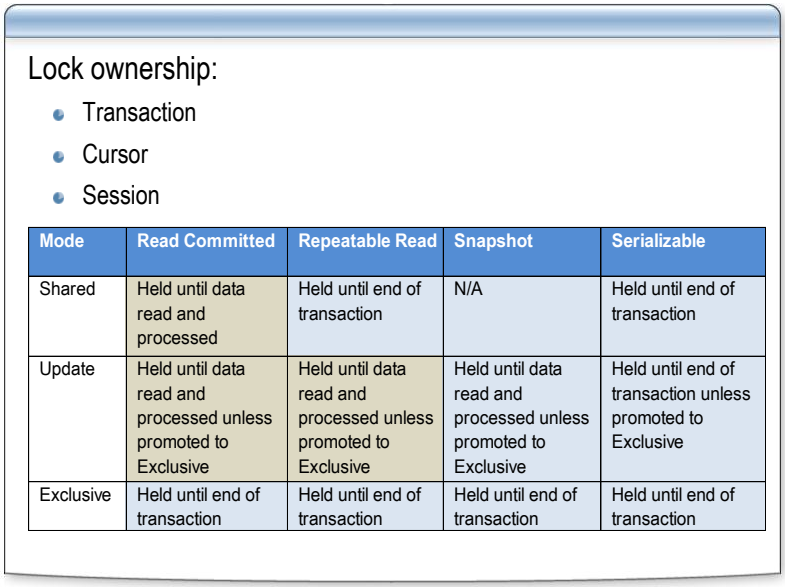
```
ALTER DATABASE DBNAME
SET READ_COMMITTED_SNAPSHOT ON
```

2. Set the transaction isolation level to READ COMMITTED. Note that this is the default transaction isolation level. Therefore, if the database option is enabled, sessions will automatically use the READ_COMMITTED_SNAPSHOT option.

To set the READ_COMMITTED_SNAPSHOT ON or OFF, there must not be any active connections to the database except for the connection running the ALTER DATABASE command. However, the database does not have to be in the single-user mode. You cannot change the state of this option when the database is OFFLINE.

You can use the sys.databases column is_read_committed_snapshot_on to determine if the option is enabled. A value of 1 indicates that the option is enabled.

# Lock Duration




| Mode | Read Committed | Repeatable Read | Snapshot | Serializable |
|---|---|---|---|---|
| Shared | Held until data read and processed | Held until end of transaction | N/A | Held until end of transaction |
| Update | Held until data read and processed unless promoted to Exclusive | Held until data read and processed unless promoted to Exclusive | Held until data read and processed unless promoted to Exclusive | Held until end of transaction unless promoted to Exclusive |
| Exclusive | Held until end of transaction | Held until end of transaction | Held until end of transaction | Held until end of transaction |

## Lock ownership

Most of the locking discussion in this topic relates to the locks owned by transactions. In addition to transactions, cursors and sessions can be owners of locks, and they both affect the duration for which locks are held.

When you use the SCROLL_LOCKS option, a cursor lock is held for every row that is fetched until the next row is fetched or the cursor is closed, regardless of the state of a transaction.

Locks owned by session are outside the scope of a transaction. The duration of these locks is bounded by the connection, and the process will continue to hold these locks until the process disconnects. A typical lock owned by session is the database (DB) lock. Shared DB lock is not held when the database is in Single user mode.

## Lock mode and transaction isolation level

For the REPEATABLE READ transaction isolation level, update locks are held until the data is read and processed, unless promoted to exclusive locks. ***Data is processed*** means that it is known if the row in question matches the search criteria. If not, then the update lock is released; otherwise, an exclusive lock is allocated to make the modification. Consider the following query:

```
dbcc traceon(3604, 1200, 1211) -- turn on lock tracing and disable escalation
go
set transaction isolation level repeatable read
begin tran
```

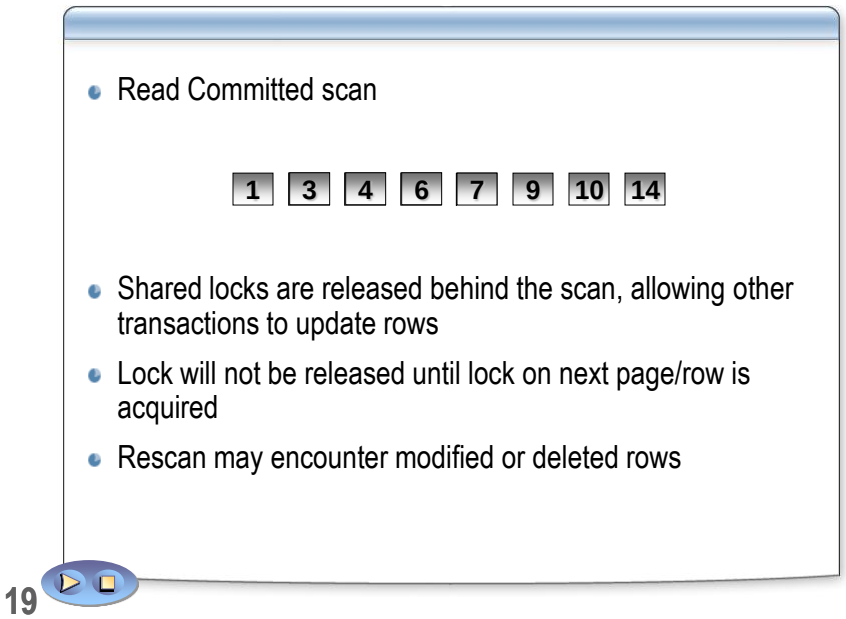
```
update Sales.SalesOrderDetail set UnitPrice = convert (real, UnitPrice) where
UnitPrice = 0.0
select * from sys.dm_tran_locks
rollback
```

In the query shown above, update locks are promoted to exclusive locks when there is a match; otherwise, the update lock is released. The sys.dm_tran_locks output shows that the SPID does not hold any update locks or shared locks at the end of the query.

Compare this update with the example shown below, where all shared locks are kept until the end of the transaction:

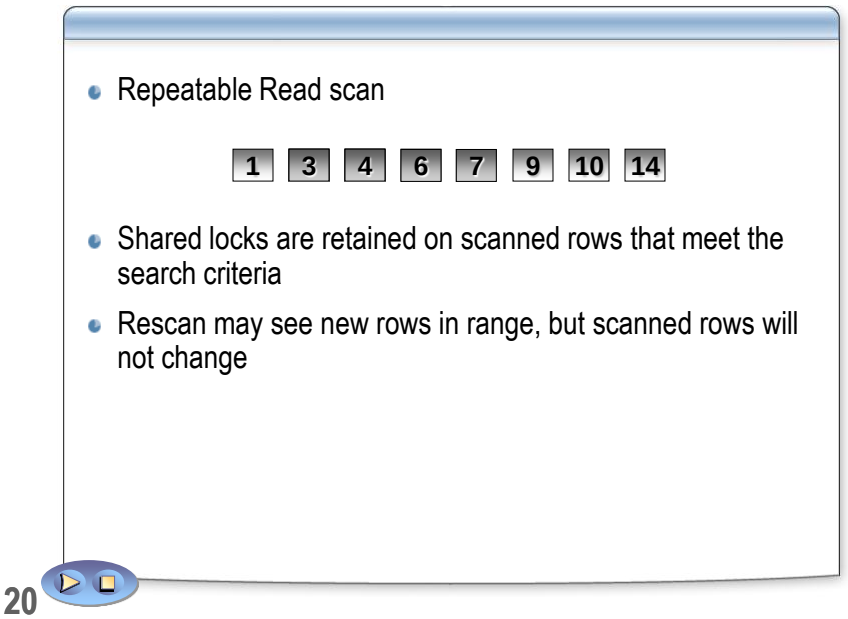
```
dbcc traceon(3604, 1200, 1211) -- turn on lock tracing and disable escalation
go
set transaction isolation level repeatable read
begin tran
select * Sales.SalesOrderDetail
select * from sys.dm_tran_locks
rollback
```

# Locking—Read Committed Scan



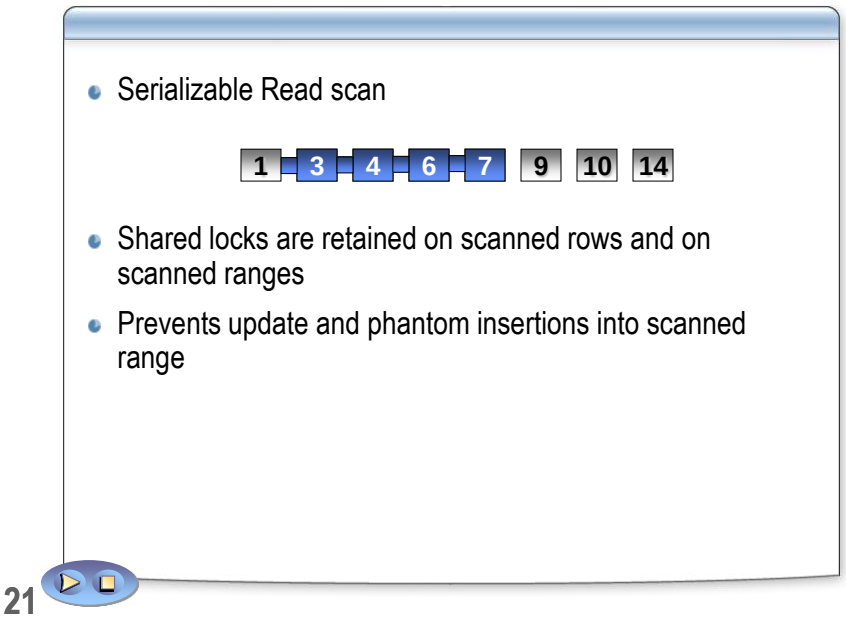

Under the Read Committed isolation level, when database pages are scanned, shared locks are held when each page is read and processed. The shared locks are then released ***behind*** the scan, which allows other transactions to update the rows. It is important to note that the shared lock currently acquired will not be released until the shared lock for the next page is successfully acquired (this is commonly known as **crabbing**). If the same pages are scanned again, rows may be modified or deleted by other transactions.

# Locking—Repeatable Read Scan



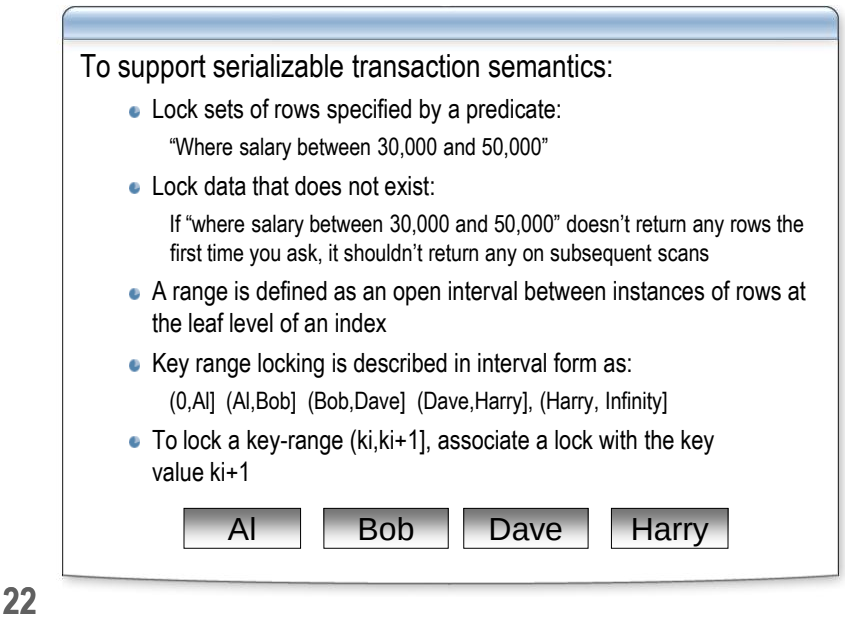

Under the Repeatable Read isolation level, when database pages are scanned, shared locks are held when each page is read and processed. SQL Server continues to hold these shared locks, thereby preventing other transactions from updating the rows. If the same pages are scanned again, previously scanned rows will not change, but new rows may be added by other transactions.

# Locking—Serializable Read Scan




Under the Serializable Read isolation level, when database pages are scanned, shared locks are held not only on rows, but also on the scanned key range. SQL Server continues to hold these shared locks until the end of the transaction. Key range locks are held, and therefore, not only are other transactions prevented from modifying the rows, but inserting any new rows is also disabled.

# Key Range Locking




To support the SERIALIZABLE transaction semantics, SQL Server needs to lock sets of rows specified by a predicate, such as:

```
WHERE salary BETWEEN 30000 AND 50000
```

***SQL Server needs to lock data that does not exist***. If no rows satisfy the WHERE condition the first time that the range is scanned, no rows should be returned on any subsequent scans either.

*Key range locks* are similar to row locks on index keys (regardless of whether they are clustered). The locks are placed on individual keys rather than at the node level.

The hash value consists of all the key components and the locator. So, for a non-clustered index over a heap, where columns c1 and c2 are indexed, the hash contains contributions from c1, c2, and the row identifier (RID). A key range lock applied to a particular key means that all keys between the value locked and the next value would be locked for all data modification.

Key range locks can lock a slightly larger range than that implied by the WHERE clause. Suppose the following SELECT statement was run in a transaction with isolation level SERIALIZABLE:

```
SELECT *
FROM members
WHERE first_name between 'Al' and 'Carl'
```

If **Al**, **Bob**, and **Dave** are index keys in the table, the first two of these would acquire key range locks. Although this would prevent anyone from inserting either **Alex** or **Ben**, it would also prevent someone from inserting **Dan**, which is not within the range of the WHERE clause.

Key-range locking prevents phantom reads. .

Key range locking ensures that the following scenarios are SERIALIZABLE:

- Range scan query
- Singleton fetch of a nonexistent row
- Delete operation
- Insert operation

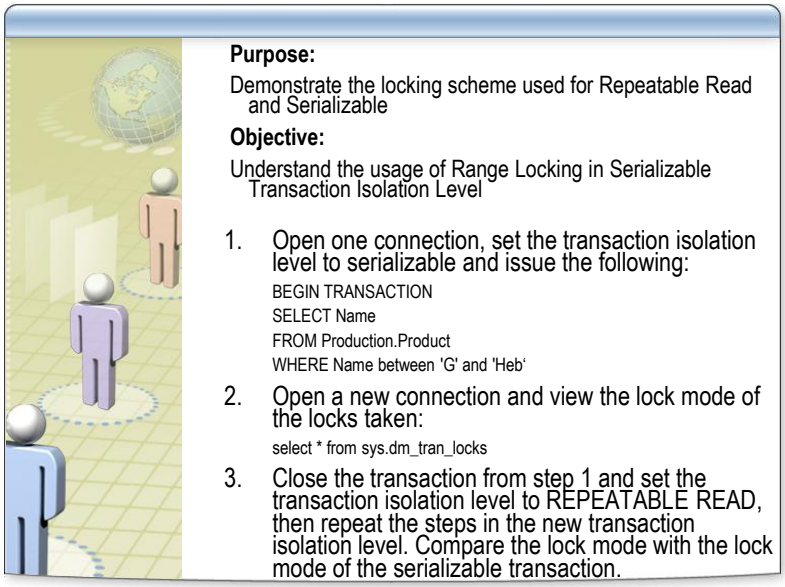
However, the following conditions must be satisfied before key range locking can occur:

- The transaction isolation level must be set to SERIALIZABLE.
- The operation performed on the data must use an index range access. Range locking is activated only when query processing (such as the optimizer) chooses an index path to access the data.
- Range Locks can occur with Default READ COMMITTED Isolation level in the following scenario when there is a fulltext index on the table.

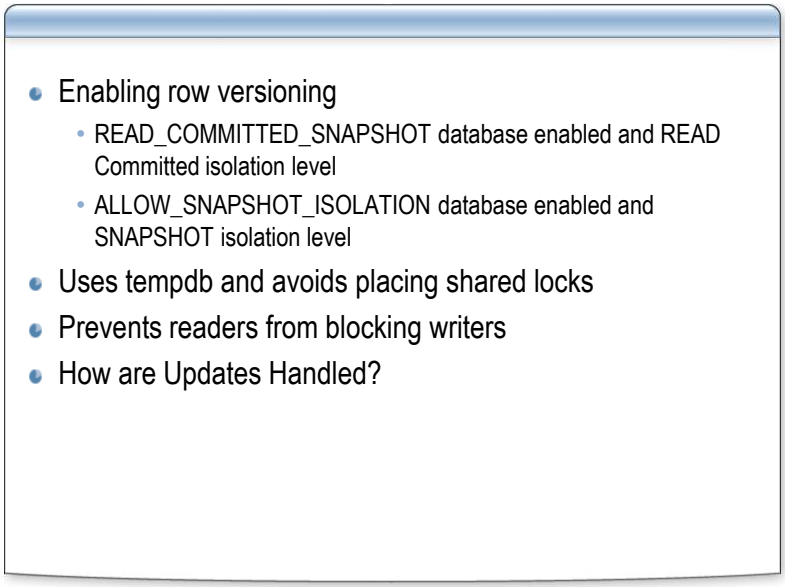
The following table describes the various key range lock modes:

| Key range lock mode | Description |
|---|---|
| RangeS_S | Shared range, shared resource lock; serializable range scan |
| RangeS_U | Shared range, update resource lock; serializable update scan |
| RangeI_N | Insert range, null resource lock; used to test ranges before inserting a new key into an index |
| RangeX_X | Exclusive range, exclusive resource lock; used when updating a key in a range |

# Demonstration 3: Range Locking



# Locking—Row Versioning




SQL Server 2005 and SQL Server 2008 provide nonlocking, nonblocking read consistency to users through snapshot isolation and read committed isolation by using ***row versioning***.

Versioning effectively starts with a *copy-on-write* mechanism that is invoked when a row is either modified or deleted. This requires that while the transaction is running, the old version of the row must be available for transactions that require an earlier, transactionally consistent state. Row versioning–based transactions can effectively *view* the consistent version of the data from these previous row versions. Row versions are stored within the version store that resides within the tempdb database.

More specifically, when a record in a table or index is modified, the new record is stamped with the *sequence_number* of the transaction that is performing the modification. The old version of the record is copied to the version store, and the new record contains a pointer to the old record in the version store. If multiple long-running transactions exist and multiple *versions* are required, records in the version store might contain pointers to even earlier versions of the row. All the earlier versions of a particular record are chained in a linked list; and in the case of long-running row versioning–based transactions, the link needs to be traversed on each access to reach the transactionally consistent version of the row. Version records must be kept in the version store only as long as there are row versioning–based queries that might be interested in them.

Before a row is changed, it is copied to a version store in **tempdb**. This is done to preserve a pre-operation version of the record, which is linked to from the post-operation

record. The post-operation record has a 14-byte tag appended to it with a transaction timestamp (TXN) and a pointer to the original record in the version store. The bytes are reclaimed the first time the row is modified after both database options are SET OFF. Adding the row versioning information can cause index page splits or the allocation of a new data page if there is not enough space available on the current page. Enough disk space should be allocated ahead of time to accommodate this requirement.

READ COMMITTED SNAPSHOT guarantees statement-level read consistency; this means that all read operations view row versions that are committed at the time that a statement started.

SNAPSHOT ISOLATION LEVEL guarantees transaction-level read consistency; this means that all read operations in a snapshot transaction view row versions that were committed at the time that the transaction started. Snapshot isolation prevents readers from blocking writers and vice versa.

## Enabling row versioning

You can enable row versioning by using ALTER DATABASE (T-SQL Syntax).

Note that when the READ_COMMITTED_SNAPSHOT database option is ON, READ_COMMITTED transactions provide statement-level read consistency by using row versioning. When the ALLOW_SNAPSHOT_ISOLATION database option is ON, SNAPSHOT transactions provide transaction-level read consistency by using row versioning.

The following statement turns ON the ALLOW_SNAPSHOT_ISOLATION database option for the Adventureworks_PTO database:

```
ALTER DATABASE Adventureworks_PTO
SET ALLOW_SNAPSHOT_ISOLATION ON
```

## Handling updates

You can handle updates by using the following commands:

- **READ_COMMITTED_SNAPSHOT**
  - Reverts from row versions to actual data to select the rows to be updated and uses update locks on the data rows selected
  - Acquires exclusive locks on actual data rows to be modified
  - Does not detect any update conflict
- **ALLOW_SNAPSHOT_ISOLATION**
  - Uses row versions to select the rows to be updated
  - Tries to acquire an exclusive lock on the actual data row to be modified, and if the data has been modified by another transaction, an update conflict occurs and the snapshot transaction is terminated

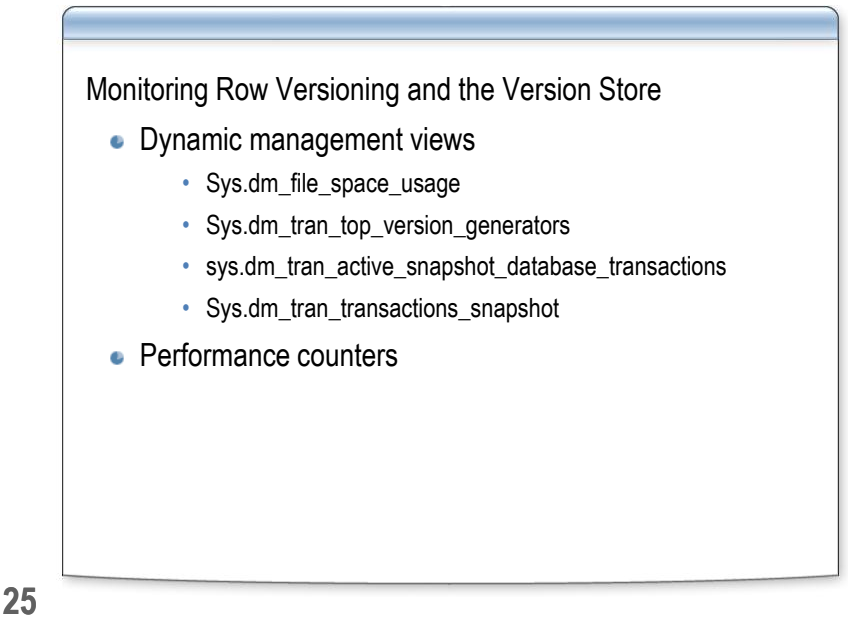
- Allows triggers, online index operations, and multiple active result sets (MARS) to continue to use versioning infrastructure, even when these options are disabled

**Note**: For more information, refer to the topic *Understanding Row Versioning-Based Isolation Levels* in SQL Server Books Online.

## Performance considerations

Row versioning-based isolation levels reduce the number of locks acquired by the transaction through the elimination of shared locks on read operations. This increases system performance by reducing the resources used to manage locks. Performance is also increased by reducing the number of times a transaction is blocked by locks acquired by other transactions. The price for this increase in performance is increased CPU and IO utilization. If the system is previously constrained by high CPU or IO and not blocking, then row versioning may have a negative impact on performance.

# Monitoring Row Versioning




You can view the current tempdb row versioning usage by querying the following DMVs:

- **sys.dm_file_space_usage.** The column version_store_reserved_page_count gives the total number of pages in the uniform extents allocated for the version store. The following query can be used to determine the space used in kilobytes:

```
SELECT (version_store_reserved_page_count * 8192)/1024 as version_store_in_kb
from sys.dm_db_file_space_usage
```

- **sys.dm_tran_top_version_generators.** This returns a virtual table for the objects that are producing the most versions in the version store.
  **sys.dm_tran_top_version_generators** returns the top 256 aggregated record lengths that are grouped by the database_id and rowset_id.
  **sys.dm_tran_top_version_generators** is based on sys**.dm_tran_version_store**

**NOTE:** sys.dm_tran_top_version_generators and sys.dm_tran_version_store are potentially very expensive functions to run, since both query the entire version store, which could be very large.

- **sys.dm_tran_active_snapshot_database_transactions.** This returns a virtual table for all active transactions in all databases within the SQL Server instance that use row versioning.

- **Sys.dm_tran_transactions_snapshot.** When a snapshot transaction starts, the Database Engine records all the transactions that are active at that time in this DMV.

Try This:

```
--Use the below mentioned query to identify the longest running batch or statement
using version store

select a.session_id,  a.elapsed_time_seconds,text
from sys.dm_tran_active_snapshot_database_transactions a
join sys.dm_exec_connections b on a.session_id= b.session_id
CROSS APPLY sys.dm_exec_sql_text(B.most_recent_sql_handle) as st
order by a.elapsed_time_seconds desc
```

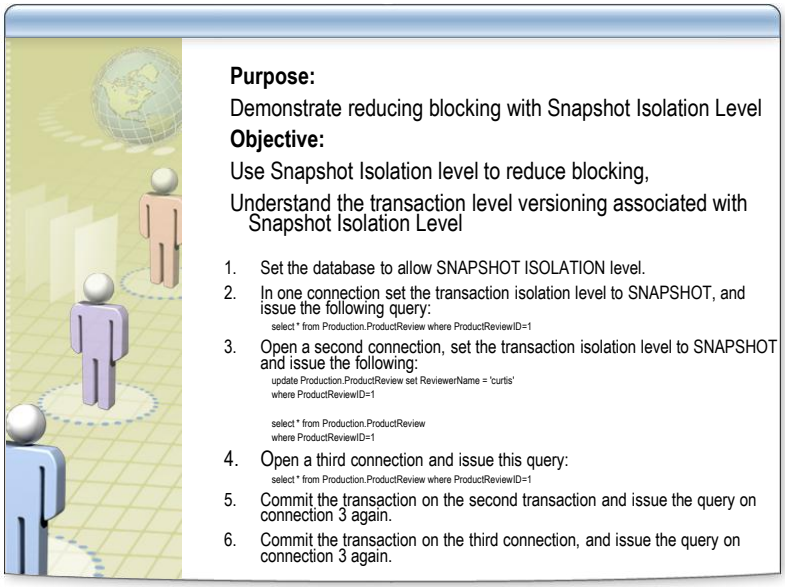
- **sys.dm_tran_current_transaction.** This is similar to **Sys.dm_tran_transactions_snapshot** in that it reports the first active transaction when the current transaction was started.

You can also monitor row versioning in perfmon. Version store counters are under the SQLServer:Transactions performance object.

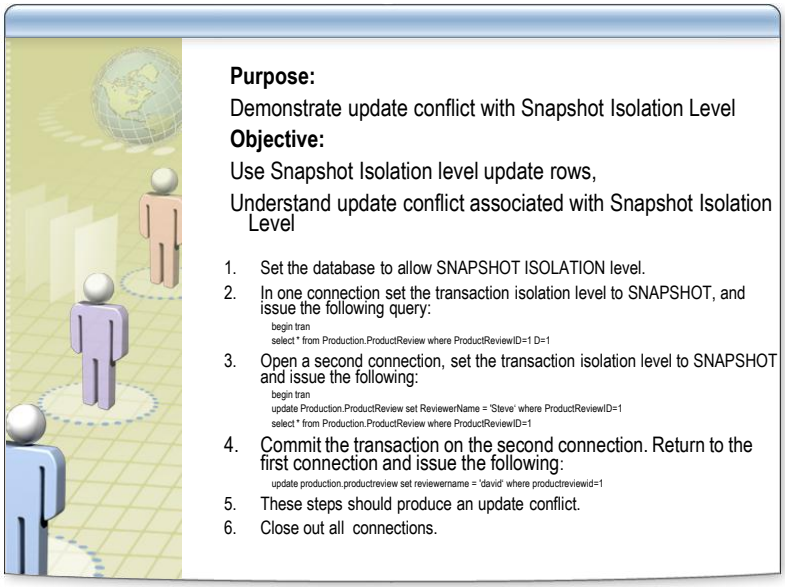
- Free Space in tempdb (KB) — Monitors the amount, in kilobytes (KB), of free space in the tempdb database. When tempdb runs out of space, the Database Engine forces the version stores to shrink. During the shrink process, the longest running transactions that have not yet generated row versions are marked as victims. A message 3967 is generated in the error log for each victim transaction. If a transaction is marked as a victim, it can no longer read the row versions in the version store. When it attempts to read row versions, message 3966 is generated and the transaction is rolled back. If the shrinking process succeeds, space becomes available in tempdb. Otherwise, tempdb runs out of space and the following occurs:
  - Write operations continue to execute but do not generate versions. An information message 3959 appears in the error log, but the transaction that writes data is not affected.
  - Transactions that attempt to access row versions that were not generated because of a tempdb full rollback terminate with an error 3958.

- Version Store Size (KB) — Monitors the size in KB of all version stores.
- Version Generation rate (KB/s) — Monitors the version generation rate in KB per second in all version stores.
- Version Cleanup rate (KB/s) — Monitors the version cleanup rate in KB per second in all version stores.
- Longest Transaction Running Time — Monitors the longest running time in seconds of any transaction using row versioning.
- Transactions — Monitors the total number of active transactions.
- Snapshot Transactions — Monitors the total number of active snapshot transactions.
- Update Snapshot Transactions — Monitors the total number of active snapshot transactions that perform update operations.
- NonSnapshot Version Transactions — Monitors the total number of active non-snapshot transactions that generate version records.

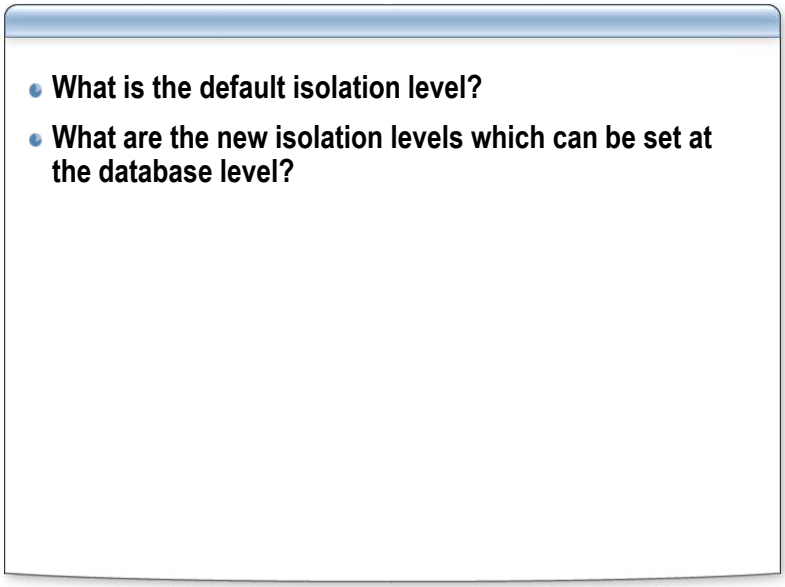
# Demonstration 4: Snapshot Isolation Level (Optional)



# Demonstration 5: Update Conflict

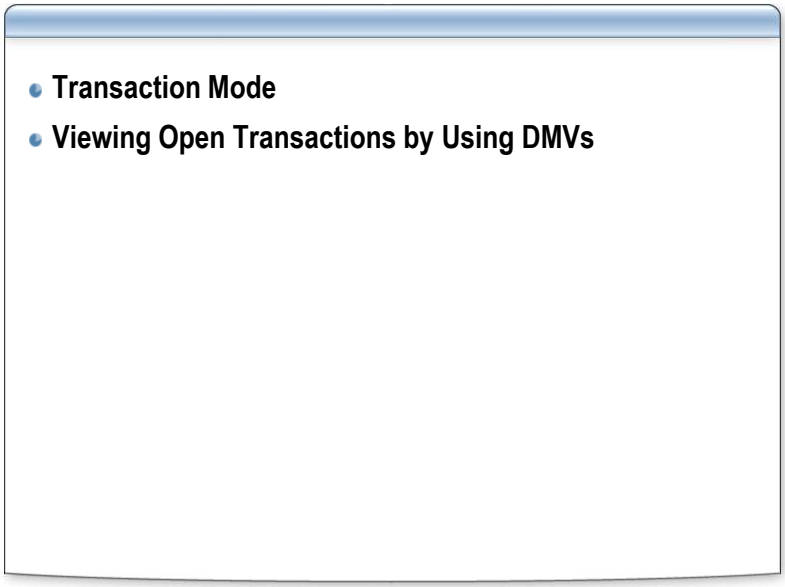


# Section 2 Review




- What is the default isolation level?
- What are the new isolation levels which can be set at the database level?

# Section 3: Transactions




## Introduction

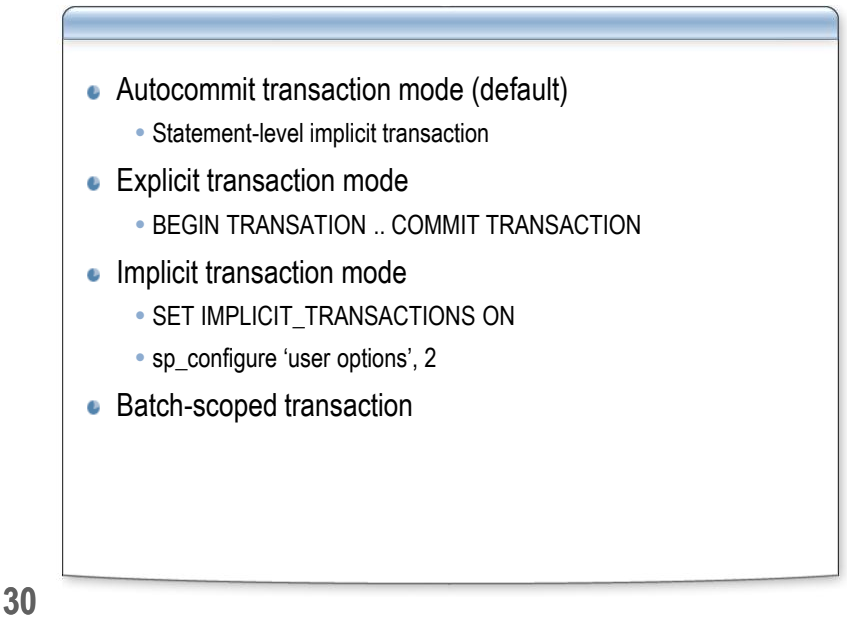
This section describes the transaction mode. This concept is useful when identifying blocking and analyzing its root cause.

## Objectives

After completing this section, you will be able to:

- Differentiate between autocommit, explicit, and implicit transaction modes.
- View open transactions by using DMVs.
- Explain how commit and rollback works in a nested transaction.
- Define statement, transaction, and batch abort.

# Transactions




A transaction is a single unit of work.

If a transaction is successful, modifications made during the transaction are committed to the database. If a transaction encounters errors and must be canceled or rolled back, modifications are erased.

SQL Server operates in the following transaction modes:

## Autocommit transaction mode (default)

Autocommit mode is the default transaction management mode of SQL Server. Every T-SQL statement, whether it is a standalone statement or part of a batch, is committed or rolled back when it completes. If a statement completes successfully, it is committed; if it encounters any error, it is rolled back. A SQL Server connection operates in autocommit mode whenever this default mode has not been overridden by either explicit or implicit transactions. Autocommit mode is also the default mode for ADO, OLE DB, ODBC, and DB-Library.

A SQL Server connection operates in autocommit mode until a BEGIN TRANSACTION statement starts an explicit transaction, or the implicit transaction mode is set ON. When the explicit transaction is committed or rolled back, or when implicit transaction mode is turned OFF, SQL Server returns to autocommit mode.

## Explicit transaction mode

An explicit transaction is a transaction that starts with a BEGIN TRANSACTION statement. An explicit transaction can contain one or more statements and must be

terminated by either a COMMIT TRANSACTION or ROLLBACK TRANSACTION statement.

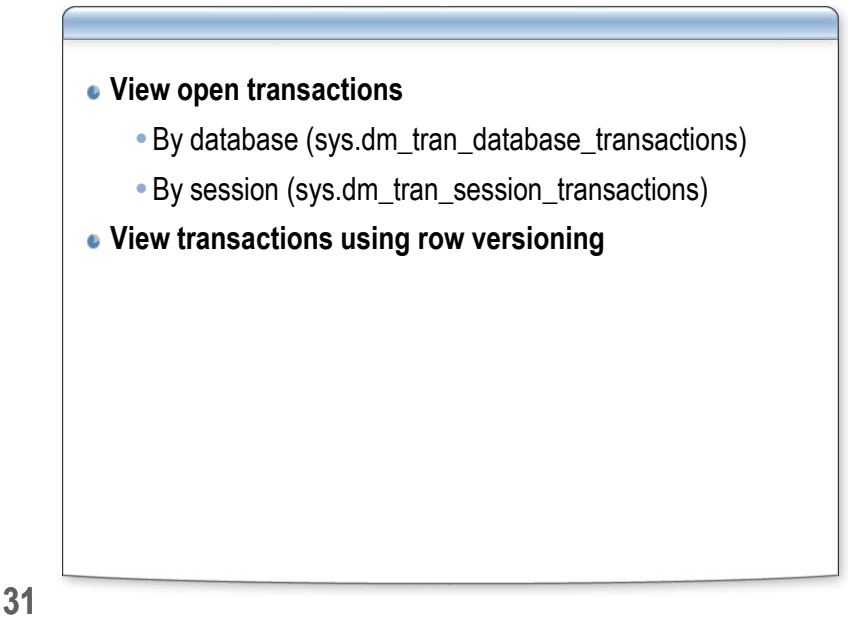
### Implicit transaction mode

SQL Server can automatically or, more precisely, implicitly start a transaction for you if the SET IMPLICIT_TRANSACTIONS ON statement is run, or if the implicit transaction option is turned on globally by running sp_configure *user options* 2. (Actually, the bit mask 0x2 must be turned ON for the user option, so you might have to perform an **OR** operation with the existing user option value.

> **Note**: For more information about turning implicit transaction ON under ODBC and OLE DB, refer to the topic *SET IMPLICIT_TRANSACTIONS (Transact-SQL)* in SQL Server 2005 Books Online Batch-Scoped Transaction.

Applicable only to MARS, a T-SQL explicit or implicit transaction that starts under a MARS session becomes a batch-scoped transaction. A batch-scoped transaction that is not committed or rolled back when a batch completes is automatically rolled back by SQL Server.

# Viewing Open Transactions Using DMVs




Applications control transactions mainly by specifying when a transaction starts and ends. You can specify the start and end of a transaction by using either T-SQL statements or database application programming interface (API) functions. The system must also be able to correctly handle errors that terminate a transaction before it completes.

## Viewing open transactions by database

You can view open transactions by using the **sys.dm_tran_database_transactions** DMV. This DMV returns the following information about currently running transactions at the database level:

- database_transaction_type
  - 1 = Read/write transaction
  - 2 = Read-only transaction
  - 3 = System transaction
- database_transaction_state
  - 1 = The transaction has not been initialized.
  - 3 = The transaction has been initialized but has not generated any log records.
  - 4 = The transaction has generated log records.
  - 5 = The transaction has been prepared.
  - 10 = The transaction has been committed.

- 11 = The transaction has been rolled back.
- 12 = The transaction is being committed. In this state, the log record is being generated, but it has not been materialized or persisted.

The query below provides a list of transactions that are rolling back:

```
select * from sys.dm_tran_database_transactions where
database_transaction_state=11
```

The query below returns the oldest transaction for a database:

```
SELECT * FROM SYS.DM_TRAN_DATABASE_TRANSACTIONS WHERE DATABASE_ID= <value> order
by database_transaction_begin_time
```

## Viewing open transactions by session

You can view information about transactions at the session level by using the **sys.dm_tran_session_transactions** DMV.

The query below generates a list of sessions with the oldest transaction:

```
select a.*, last_request_start_time from sys.dm_tran_session_transactions a join
sys.dm_exec_sessions b on a.session_id = b.session_id order by
last_request_start_time
```

The **sys.dm_tran_active_transactions** DMV returns a list of transactions.

## Viewing transactions by using row versioning

The **sys.dm_tran_active_snapshot_database_transactions** DMV returns rows for all active transactions in all snapshot-enabled databases.

The query below returns the 10 longest transactions that are using the version store:

```
SELECT TOP 10 transaction_id FROM
sys.dm_tran_active_snapshot_database_transactions ORDER BY elapsed_time_seconds
```

The **sys.dm_tran_current_snapshot** DMV returns a list of transactions that are active when each snapshot transaction starts.

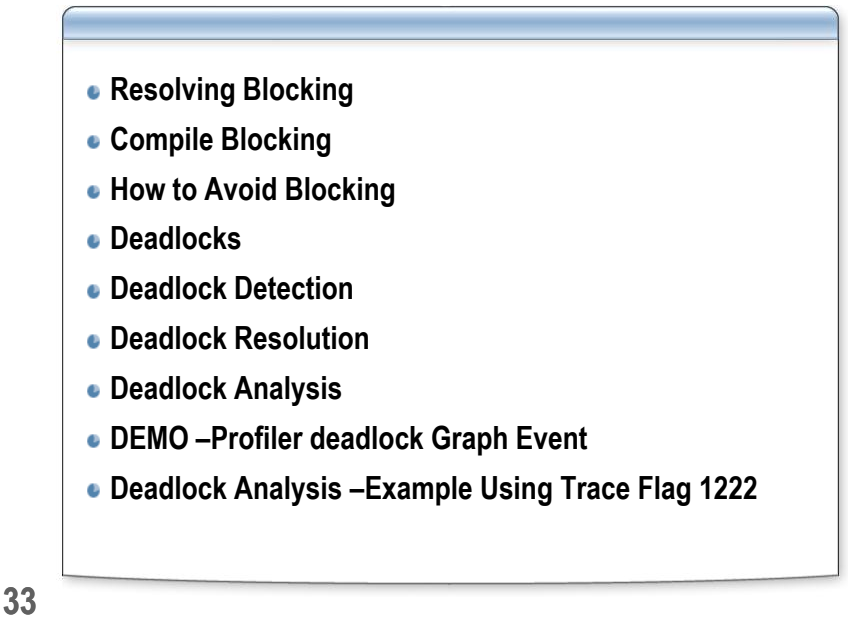
The **sys.dm_tran_version_store** DMV displays all version records in the version store. Querying this DMV is inefficient to run because it queries the entire version store, which can be very large.

# Section 3 Review

- What is the default transaction mode in SQL Server?
- What could be the impact of using the sys.dm_tran_version_store DMV?

32

# Section 4: Blocking and Deadlocking




## Introduction

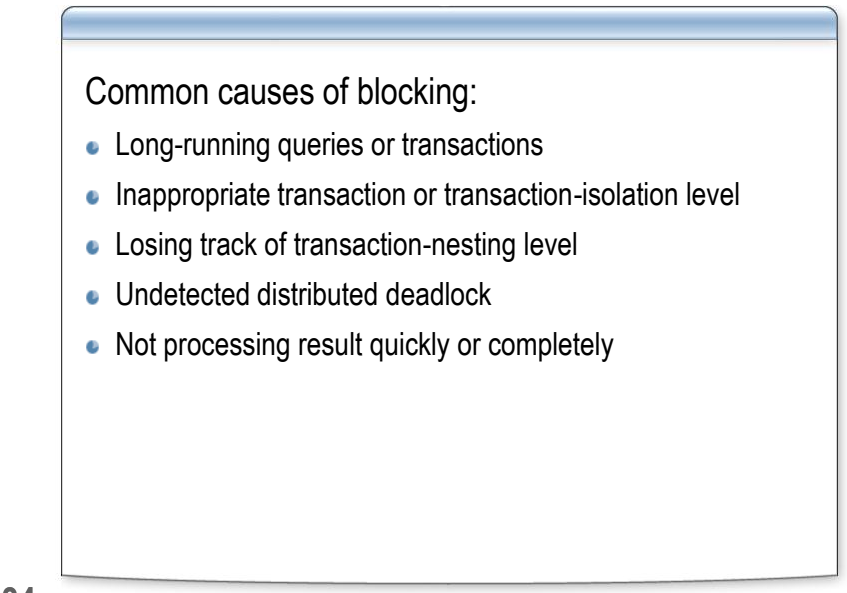
Users modifying data can affect other users who are reading or modifying the same data at the same time. In some scenarios, this could lead to blocking or deadlocking.

## Objectives

After completing this section, you will be able to:

- Resolve issues related to blocking, such as long-running queries, inappropriate transaction isolation levels, and slow or incomplete results.
- Troubleshoot issues related to compile blocking.
- Avoid common blocking issues.
- Define deadlocks.
- Explain how SQL Server resolves deadlocks.
- Analyze deadlocks by using Trace Flag 1222.

# Resolving Blocking




Users who access a resource at the same time are said to be accessing the resource concurrently. Concurrent data access requires mechanisms to prevent adverse effects when multiple users try to modify resources that are actively being used by other users.

The different levels of concurrency control have different side effects.

## Blocking caused by long-running queries

After you have identified the blocking session, the row associated with that session or request in **sys.dm_exec_sessions** or **sys.dm_exec_requests** will show the following column values:

| Column | Value |
|---|---|
| status | Runnable. The request is running and temporarily scheduled out. |
| wait_type | If the request is blocked, this column returns the type of wait. |

Additionally, you can monitor the blocking session through the SQL Server Profiler. Monitor by using the following profiler events:

- **SQL:StmtCompleted (SQL:BatchCompleted)**
- **SP:StmtCompleted (RPC:Completed)**

The duration column in these events will have a relatively high value.

Other supporting information includes high values for **sys.dm_exec_requests** columns, including the **cpu_time**, **reads**, **writes**, **logical_reads**, and **granted_query_memory**

columns. A high value for any of these columns indicates that the request has been utilizing a large amount of resources to process queries.

To resolve blocking caused by long-running queries, you can:

- Improve the performance of the long-running queries.
- Run the queries somewhere else or run them at a different time to minimize their impact on concurrency.
- Consider to implement Row Versioning.

### Blocking caused by inappropriate transaction or transaction isolation level

After you have identified the blocking session and request, the rows associated with them in **sys.dm_exec_sessions** and **sys.dm_exec_requests** will show the following column values:

| Column | Value |
|---|---|
| status | Runnable. The request is running and temporarily scheduled out. |
| wait_type | If the request is blocked, this column returns the type of wait. |
| open_transaction_count | This column specifies the number of transactions that are open for this request. If this is greater than 0, it indicates that the process is in transaction. |

### Losing track of transaction-nesting level

After you have identified the blocking session and request, the rows associated with them in **sys.dm_exec_sessions** and **sys.dm_exec_requests** will show the following values:

| Column | Value |
|---|---|
| status | Sleeping. There is no work to be done. |
| wait_type | NULL |
| open_transaction_count | Number of transactions that are open for this request will be 0. |

Additionally, you can monitor the blocking session and request through the SQL Server Profiler. Monitor by using the following profiler events:

- **Attention** event
- **Exception** event
- Or both

An orphaned transaction is usually caused by query cancellation or timeout without a rollback. Timeouts and most errors do not roll back any active transactions. It is the responsibility of the application to explicitly request a rollback in the events of an error or timeout.

## Blocking caused by undetected distributed deadlock

After you have identified the blocking session and request, the rows associated with them in **sys.dm_exec_sessions** and **sys.dm_exec_requests** will show the following column values:

| Column | Value |
|---|---|
| status | Sleeping. There is no work to be done. |
| wait_type | NULL |
| open_transaction_count | This column specifies the number of transactions that are open for this request. If this is greater than 0, it indicates that the process is in transaction. |

Undetected deadlocks are difficult to find and troubleshoot. Application developers must be aware of them and must code applications to handle these cases appropriately.

> **Note**: You will learn more about resolution for distributed deadlock in the *Deadlock Section* later in the workbook.

## Blocking caused by slow or incomplete result processing

After you have identified the blocking SPID, the row associated with that SPID in sys.sysprocesses will show the following column values:

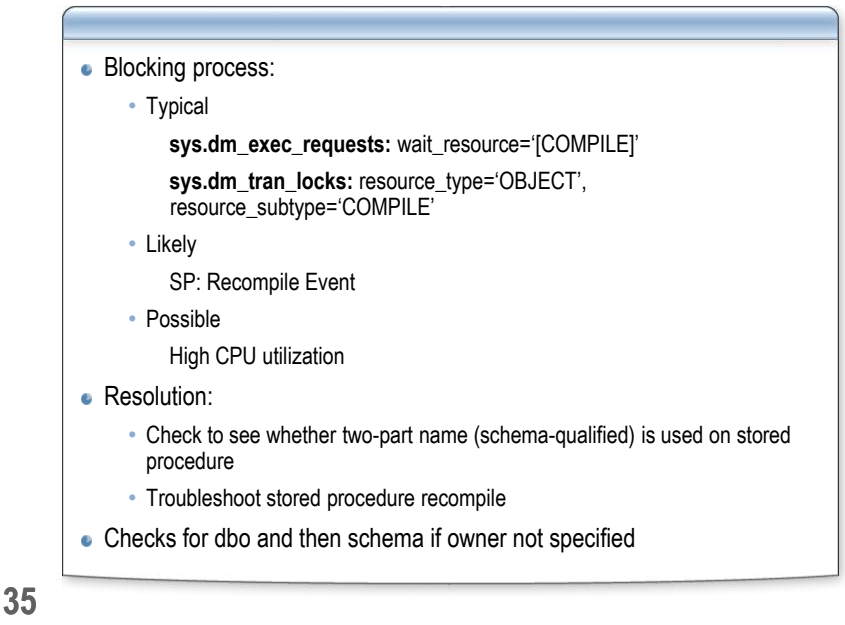
| Column | Value |
|---|---|
| status | Runnable. The request is running and temporarily scheduled out. |
| wait_type | ASYNC_NETWORK_IO (Occurs on network writes when the task is blocked behind the network. Verify that the client is processing data from the server.) |
| open_transaction_count | This column specifies the number of transactions that are open for this request. If this is greater than 0, it indicates that the process is in transaction. |

The *runnable* status shows that the query is still running, the **wait_type** = ASYNC_NETWORK_IO indicates that the query is waiting on the network resource, and **open_transaction_count** >= 0 indicates that the process may or may not be in a transaction.

SQL Server only caches one network buffer (4 K by default) of rows while waiting for the client to signal that it is ready for more data. If an application is slow in fetching rows produced by a query, SQL Server holds shared locks so that the engine can resume at the scan point when the client is ready for more. If you are using an application that transparently submits SQL statements to the server, the application must fetch all result rows. If it does not (and if it cannot be configured to do so), you may be unable to resolve the blocking problem.

To fully resolve the issue, you must rewrite the application to fetch all rows of the result to completion.

# Compile Blocking




Compile blocking appears as blocked requests waiting on a COMPILE resource in **sys.dm_exec_requests.wait_resource** and **sys.dm_tran_locks**. In addition, there may be a large number of SQL Profiler **SP:Recompile** events in the trace. Compile blocking will usually cause ***rolling blocking***. Rolling blocking means that the blocked requests at the end of the chain may have to wait a long time, but no single blocker remains at the head of the chain for long. Other symptoms may include high CPU utilization because compilation is CPU-intensive.

> **Important:** SQL Server needs to acquire the compile lock to compile the procedure for the first time.

In SQL Server, only one copy of a stored procedure plan is generally cached at a time, so stored procedure compilation attempts are serialized. If a commonly used stored procedure must be frequently recompiled (for example, cursor on a temp table), it is possible for this stored procedure to cause blocking as one request waits on another to finish compilation.

In this situation, you can do one of the following:

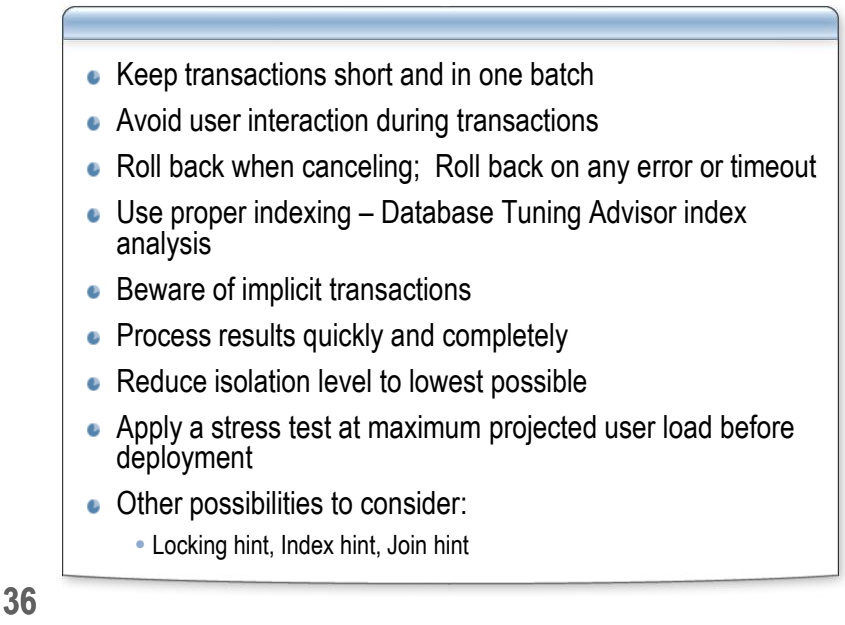
- Reduce recompile frequency.
- Owner-qualify stored procedures. Compile locks can be held for longer than is necessary if the name of the procedure is not fully qualified with the database name and the owner name, and the user is not the owner of the procedure.

During name resolution, when the owner of the procedure is not specified, SQL Server first looks for a procedure owned by that user. If SQL Server cannot find this procedure, it then checks for an object owned by Database Owner (dbo). For example, if the user Joe runs a procedure Proc1 owned by dbo, during name resolution, SQL Server will first check for the procedure Joe.Proc1. If SQL Server does not find this procedure, it will then look for the procedure dbo.Proc1.

During the process of looking for a procedure where the owner name is not qualified and the initial lookup by user fails, SQL Server acquires an exclusive compile lock on the procedure. A second lookup is made in cache to see if a procedure owned by dbo can be found, while this lock is in place. If a usable copy of the compile plan is found in cache, the procedure is not recompiled.

It is the lack of owner qualification that requires the exclusive compile lock be placed before determining if the plan can be reused. Acquiring the lock, the second lookup, and other necessary work can introduce a delay that results in the compile lock being held longer, resulting in a blocking lock. Even if the information gathered shows that the users are not blocking each other with compile locks, lack of owner qualification can introduce delays in execution and unnecessarily high CPU utilization.

# How to Avoid Blocking




SQL Server is a transaction-oriented relational database management system (RDBMS) that is often used in highly concurrent environments with many simultaneous users. It maintains transactional integrity and database consistency by using locks.

An unavoidable characteristic of any lock-based concurrent system is that blocking may occur under some conditions. Blocking happens when one connection holds a lock and a second connection wants a conflicting lock type. This forces the second connection to either wait or block the first one.

The following sections describe some preventive measures to avoid blocking problems.

## Short transactions

Avoid user interaction in transactions, and keep transactions as short as possible and in a single batch.

## Handling cancellations and errors with care

If the application (or any stored procedures that the application invokes) starts any transactions, the application is responsible for rolling back any active transactions (IF @@TRANCOUNT>0 ROLLBACK TRAN) when any error or query timeout occurs. The application should not assume that an arbitrary error would abort the transaction. A query timeout never aborts transactions automatically.

> **Important:** It is especially important for an application to handle transactions correctly when connection pooling is enabled because a connection with an orphaned transaction can be returned to the pool.

## Proper indexing

On a table where inserts and updates are common, create a clustered index. Inserts into a heap should not cause a hotspot issue, but they can be much slower than inserts into the same table with a clustered index due to contention on Page Free Space (PFS) (longer transactions = greater chance of blocking). In general, the best column for a clustered index is often an *identity column*.

Avoid placing a clustered index on columns that are frequently updated. Updates to clustered index key columns will require locks on the clustered index, data pages (to move the row), and all non-clustered indexes (because non-clustered indexes point to rows via the clustered index key). Consider using a clustered index on an identity column or a column with similar properties, such as small data values or constantly increasing values with very infrequent modifications.

Analyze indexing for possible improvements. Poor indexes mean more pages visited, more locks acquired, and longer-running transactions (all these contribute to blocking, and in general, may increase the chances of deadlocking). Use the Database Engine Tuning Advisor (DTA) on related queries.

Remove indexes that have extremely low selectivity. For example, if an indexed column on a large table only has 10 distinct values, a KEY lock on the index can effectively prevent updates to this column for 10 percent of the rows in the table.

## Beware of implicit transactions

With implicit transactions on, every time you make a modification, you get a transaction started automatically that you must commit yourself. Many modifications do not need to be part of a larger transaction. The improper use of implicit transaction tends to end up with two undesirable outcomes:

- The developer forgets about the open transaction and leaves it open across modifications, which spans much more work than required by the business needs. Longer transactions than necessary mean more locks held, greater chance of deadlock or blocking, etc.
- The developer does the right thing and commits the implicit transaction after every unit of work. But this requires an extra round-trip to the server to send the COMMIT command, while autocommit (the opposite of implicit transactions) could have accomplished the same thing with a single query and a single round-trip. Also, because implicit transactions cause every command to open a transaction, the danger due to the coding mistakes that lead to orphaned transactions is increased.

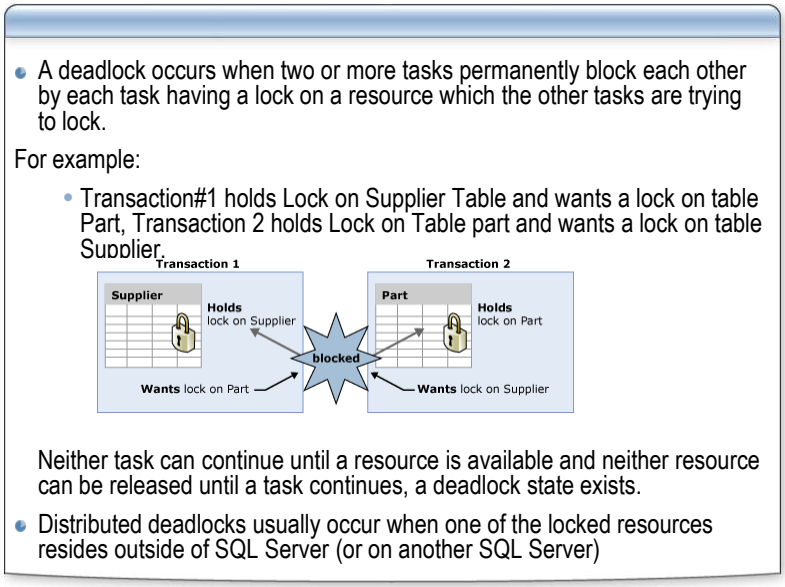
Therefore, it is important to be careful when using implicit transaction to avoid losing track of transaction nesting.

## Use as low an isolation level as possible

Higher transaction isolation levels (such as SERIALIZABLE) hold more locks for a longer period of time. Microsoft Transaction Server always sets the transaction isolation level to SERIALIZABLE. You can change this either by using the SET TRANSACTION

ISOLATION LEVEL command or on a query-by-query basis with a READ COMMITTED query hint.

# Deadlocks




## What is a deadlock?

A deadlock exists when the following four conditions exist simultaneously:

1. No pre-emption
2. Incremental allocation of resources (Locks are acquired as needed, rather than acquiring them all before beginning the transaction)
3. Incompatible use of resources (Here, compatibility refers to the one defined in our *lock compatibility table*)
4. Cycle When the lock monitor initiates deadlock search for a particular thread, it identifies the resource on which the thread is waiting. The lock monitor then finds the owner(s) for that particular resource and recursively continues the deadlock search for those threads until it finds a cycle. A cycle identified in this manner forms a deadlock.

Deadlocking is not specific to SQL Server or even to relational databases. Any system that must manage shared resources among multiple users, threads, or processes may be susceptible to deadlocks.

Each user session might have one or more tasks running on its behalf where each task might acquire or wait to acquire a variety of resources. The following types of resources can cause blocking that could result in a deadlock:

- Locks
- Worker threads
- Memory
- Parallel query execution-related resources
- Multiple Active Result Sets (MARS) resources

SQL Server always resolves deadlocking by choosing a victim and therefore, breaking the cycle. Most often, you can address deadlocking by using methods such as a ***nolock*** hint To address a deadlocking situation with item 2 in the list above (worker threads), you first need to understand what is being locked and in what order.

To understand this better, consider the following examples:

- You might address a deadlock between two tables by making sure that you access the two tables in the same order. A key-RID deadlock caused by a SELECT and UPDATE or two updates may be addressed either by converting a non-clustered index to a clustered index, where practical, or by using a *lock earlier* strategy in the transaction.
- A deadlock can also be caused by two updates that are expensive enough that they go parallel. Therefore, you do not have a guarantee that the two rows in the same table are accessed in the same order. You may resolve such a deadlock by adding an appropriate index so that the update can be performed more efficiently, without getting a parallel execution plan.
- A deadlock involving two rows on the same table with an application that uses positioned updates may be resolved by the use of an ***order by***.

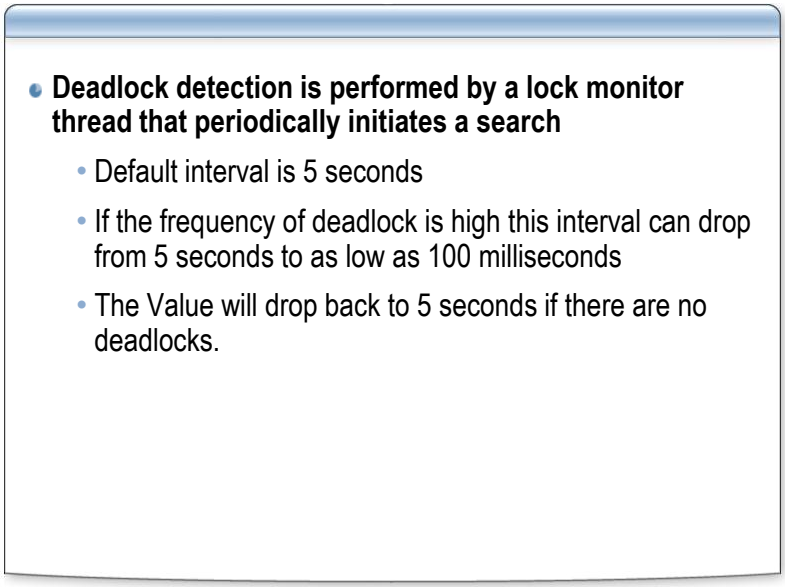
## Distributed deadlock

Simple distributed deadlocks usually occur when one of the locked resources resides outside of SQL Server. For example, application A holds a lock on a SQL Server resource (table, page, key, or row) and is waiting to get exclusive access to a particular file. Application B holds an exclusive lock on this file, but is blocked waiting on a SQL Server resource locked by application A.

## Resolving distributed deadlocks

In the scenario described above, SQL Server does not have any knowledge of the external file or the dependency of the application on it. Prior to SQL Server 2000, this situation would have to be managed entirely by the application via lock or query timeouts, and so on. SQL Server 2005 introduces two new stored procedures—**sp_getapplock** and **sp_releaseapplock**—that can be called by the application to let SQL Server know when it has acquired a particular type of lock on an external object. If all applications sharing the external resource are written to use these stored procedures, then SQL Server can detect the deadlock situations involving non-SQL objects and inform the application when it has encountered a deadlock.

# Deadlock Detection

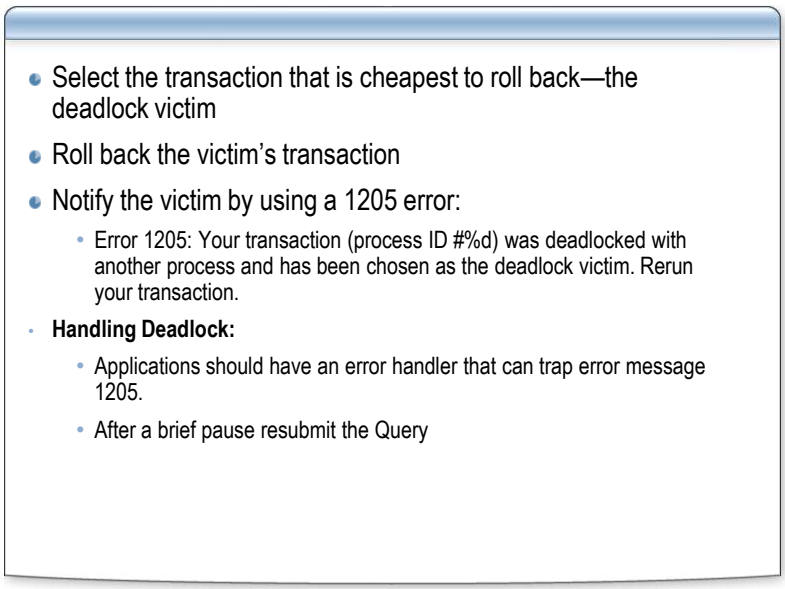



All of the resources listed in the previous topic participate in the Database Engine deadlock detection scheme. Deadlock detection is performed by a lock monitor thread that periodically initiates a search through all the tasks in an instance of the Database Engine. When the lock monitor initiates deadlock search for a particular thread, it identifies the resource on which the thread is waiting.

The following points describe the search process:

- The default interval is 5 seconds.
- If the lock monitor thread finds deadlocks, the deadlock detection interval will drop from 5 seconds to as low as 100 milliseconds depending on the frequency of deadlocks.
- If the lock monitor thread stops finding deadlocks, the Database Engine increases the intervals between searches to 5 seconds.

# Deadlock Resolution




## How is a deadlock resolved?

SQL Server picks one of the connections as a deadlock victim. The victim is chosen based on either the least-expensive transaction (calculated by using the number and size of the log records) to roll back and according to the SET DEADLOCK_PRIORITY specified. The operation looks like this:

```
SET DEADLOCK_PRIORITY { LOW | NORMAL | HIGH | <numeric-priority> |
@deadlock_var | @deadlock_intvar }
```

The victim's transaction is rolled back, held locks are released, and then SQL Server sends error 1205 to the victim's client application to notify it that it was chosen as a victim. The other process can then obtain access to the resource that it was waiting on and continue.

Error 1205 appears as follows:

```
Error 1205: Your transaction (process ID #%d) was deadlocked with another process
and has been chosen as the deadlock victim. Rerun your transaction.
```

## Symptoms of deadlocking

Error 1205 is not normally written to the SQL Server error log. Unfortunately, you cannot use sp_altermessage to cause 1205 to be written to the error log.

If the client application does not capture and display error 1205, you can identify a deadlock by observing some of the other symptoms such as:

- Mysteriously canceled queries when using certain features of an application

- Excessive blocking

In addition, lock contention increases the chances of a deadlock.

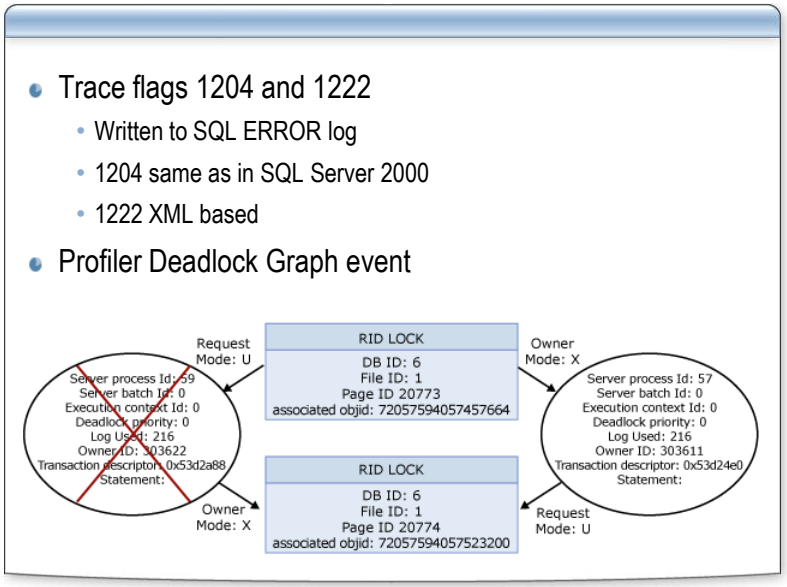
## Handling deadLocks

Applications should have an error handler that can trap error 1205. If an application does not trap the error, the application can proceed unaware that its transaction has been rolled back and errors can occur.

Implementing an error handler that traps error 1205 allows an application to handle the deadlock situation and take remedial action.

The application should pause briefly before resubmitting its query. This gives the other transaction involved in the deadlock a chance to complete and release its locks that formed part of the deadlock cycle.

# Deadlock Analysis




To view deadlock information, the Database Engine provides monitoring tools in the form of two *Trace Flags*, and the *deadlock graph event* in SQL Server Profiler.

## Trace Flags

- Trace Flag 1204. It is focused on the nodes involved in the deadlock. Each node has a dedicated section, and the final section describes the deadlock victim.

Example of Trace Flag 1204:

```
Deadlock encountered .... Printing deadlock information
Wait-for graph
Node:1
RID: 6:1:20789:0               CleanCnt:3 Mode:X Flags: 0x2
 Grant List 0:
   Owner:0x0315D6A0 Mode: X
     Flg:0x0 Ref:0 Life:02000000 SPID:55 ECID:0 XactLockInfo: 0x04D9E27C
   SPID: 55 ECID: 0 Statement Type: UPDATE Line #: 6
   Input Buf: Language Event:
BEGIN TRANSACTION
EXEC usp_p2
 Requested By:
   ResType:LockOwner Stype:'OR'Xdes:0x03A3DAD0
     Mode: U SPID:54 BatchID:0 ECID:0 TaskProxy:(0x04976374) Value:0x315d200
Cost:(0/868)

Node:2
KEY: 6:72057594057457664 (350007a4d329) CleanCnt:2 Mode:X Flags: 0x0
 Grant List 0:
   Owner:0x0315D140 Mode: X
     Flg:0x0 Ref:0 Life:02000000 SPID:54 ECID:0 XactLockInfo: 0x03A3DAF4
```

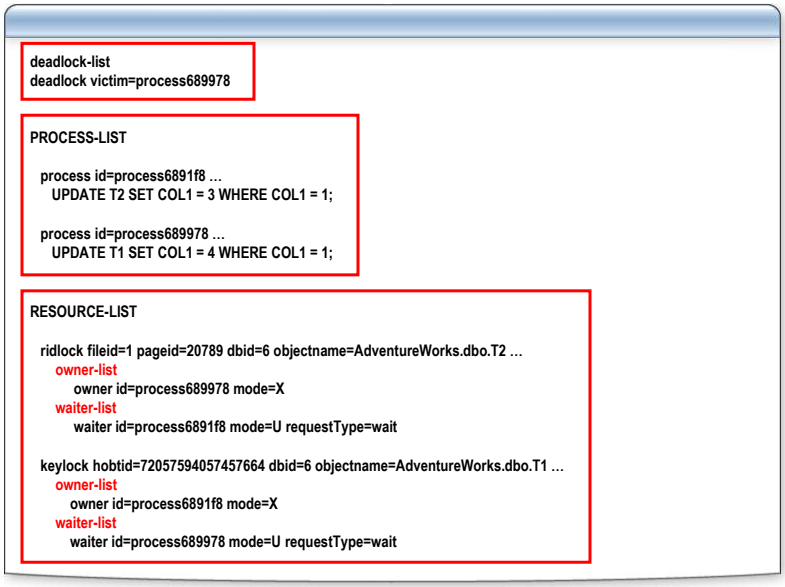
```
   SPID: 54 ECID: 0 Statement Type: UPDATE Line #: 6
   Input Buf: Language Event:
     BEGIN TRANSACTION
EXEC usp_p1
 Requested By:
   ResType:LockOwner Stype:'OR'Xdes:0x04D9E258
     Mode: U SPID:55 BatchID:0 ECID:0 TaskProxy:(0x0475E374) Value:0x315d4a0
Cost:(0/380)
Victim Resource Owner:
ResType:LockOwner Stype:'OR'Xdes:0x04D9E258
Mode: U SPID:55 BatchID:0 ECID:0 TaskProxy:(0x0475E374) Value:0x315d4a0
Cost:(0/380)
```

- Trace Flag 1222. It returns information in an XML-like format that does not conform to an XML Schema Definition (XSD) schema. This format has three major sections:
  - The first section declares the deadlock victim.
  - The second section describes each process involved in the deadlock.
  - The third section describes the resources that are synonymous with nodes in Trace Flag 1204.

### Profiler deadlock graph event

This is an event in SQL Server Profiler that presents a graphical depiction of the tasks and resources involved in a deadlock.

# Deadlock Analysis—Example Using Trace Flag 1222




The output from Trace Flag 1222 is divided into three parts:

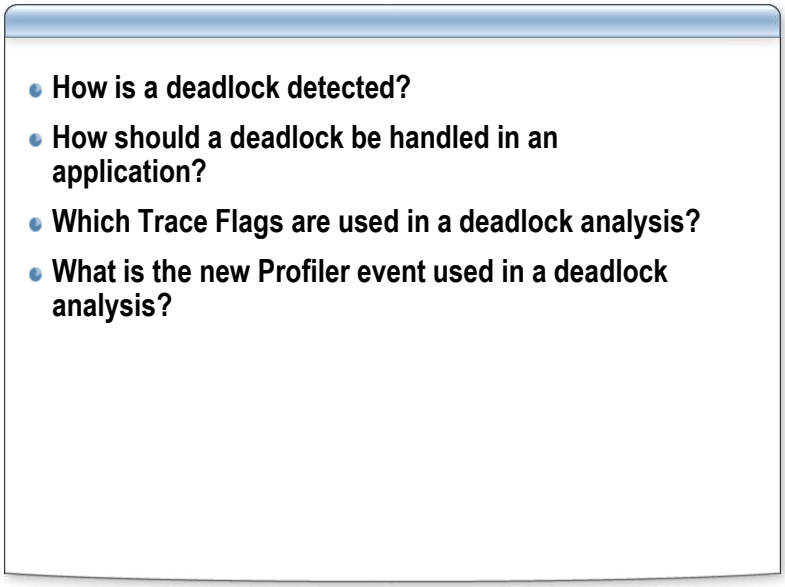
- **Deadlock victim** contains the session that has been broken.
- **Process list** shows all outstanding sessions involved in the deadlock.
- **Resource list** contains a list of resources involved in the deadlock.

The following example shows the output when Trace Flag 1222 is turned ON:

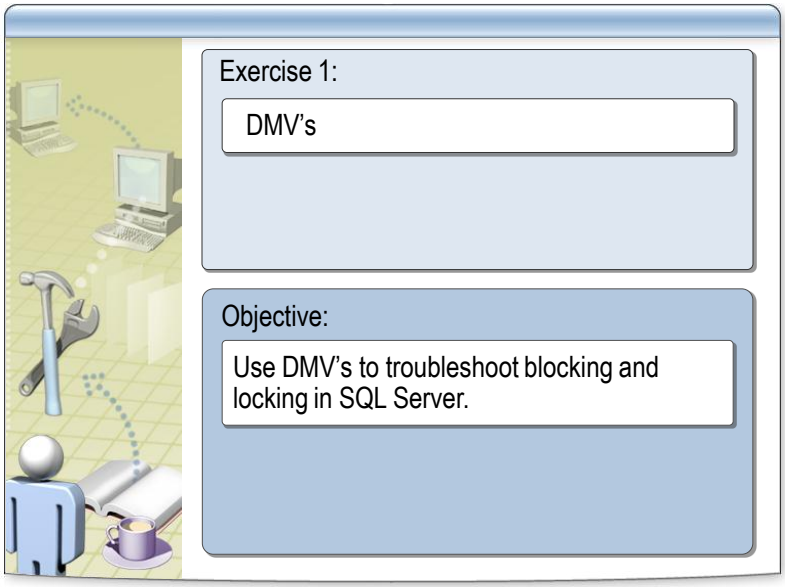
```
deadlock-list
 deadlock victim=process689978
  process-list
   process id=process6891f8 taskpriority=0 logused=868
   waitresource=RID: 6:1:20789:0 waittime=1359 ownerId=310444
   transactionname=user_transaction
   lasttranstarted=2005-09-05T11:22:42.733 XDES=0x3a3dad0
   lockMode=U schedulerid=1 kpid=1952 status=suspended spid=54
   sbid=0 ecid=0 priority=0 transcount=2
   lastbatchstarted=2005-09-05T11:22:42.733
   lastbatchcompleted=2005-09-05T11:22:42.733
   clientapp=Microsoft SQL Server Management Studio - Query
   hostname=TEST_SERVER hostpid=2216 loginname=DOMAIN\user
   isolationlevel=read committed (2) xactid=310444 currentdb=6
   lockTimeout=4294967295 clientoption1=671090784 clientoption2=390200
    executionStack
     frame procname=AdventureWorks.dbo.usp_p1 line=6 stmtstart=202
     sqlhandle=0x0300060013e6446b027cbb00c6960000010000000000000
     UPDATE T2 SET COL1 = 3 WHERE COL1 = 1;
     frame procname=adhoc line=3 stmtstart=44
     sqlhandle=0x01000600856aa70f503b8104000000000000000000000000
```

```
     EXEC usp_p1
    inputbuf
      BEGIN TRANSACTION
EXEC usp_p1
   process id=process689978 taskpriority=0 logused=380
   waitresource=KEY: 6:72057594057457664 (350007a4d329)
   waittime=5015 ownerId=310462 transactionname=user_transaction
   lasttranstarted=2005-09-05T11:22:44.077 XDES=0x4d9e258 lockMode=U
   schedulerid=1 kpid=3024 status=suspended spid=55 sbid=0 ecid=0
   priority=0 transcount=2 lastbatchstarted=2005-09-05T11:22:44.077
   lastbatchcompleted=2005-09-05T11:22:44.077
   clientapp=Microsoft SQL Server Management Studio - Query
   hostname=TEST_SERVER hostpid=2216 loginname=DOMAIN\user
   isolationlevel=read committed (2) xactid=310462 currentdb=6
   lockTimeout=4294967295 clientoption1=671090784 clientoption2=390200
    executionStack
     frame procname=AdventureWorks.dbo.usp_p2 line=6 stmtstart=200
     sqlhandle=0x030006004c0a396c027cbb00c69600000100000000000000
     UPDATE T1 SET COL1 = 4 WHERE COL1 = 1;
     frame procname=adhoc line=3 stmtstart=44
     sqlhandle=0x01000600d688e709b85f89040000000000000000000000000
     EXEC usp_p2
    inputbuf
      BEGIN TRANSACTION
EXEC usp_p2
  resource-list
   ridlock fileid=1 pageid=20789 dbid=6 objectname=AdventureWorks.dbo.T2
   id=lock3136940 mode=X associatedObjectId=72057594057392128
    owner-list
     owner id=process689978 mode=X
    waiter-list
     waiter id=process6891f8 mode=U requestType=wait
   keylock hobtid=72057594057457664 dbid=6 objectname=AdventureWorks.dbo.T1
   indexname=nci_T1_COL1 id=lock3136fc0 mode=X
   associatedObjectId=72057594057457664
    owner-list
     owner id=process6891f8 mode=X
    waiter-list
     waiter id=process689978 mode=U requestType=wait
```

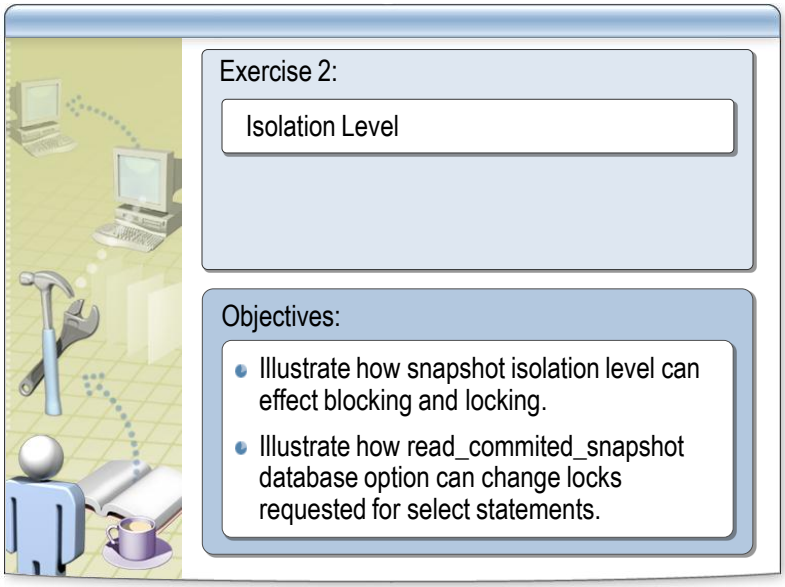
# Section 4 Review



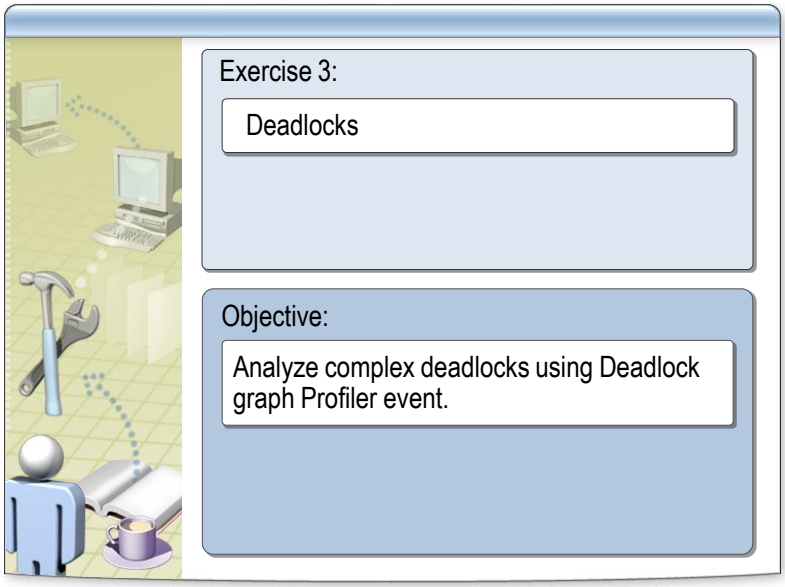
# Lab 1: Exercise 1



# Lab 1: Exercise 2

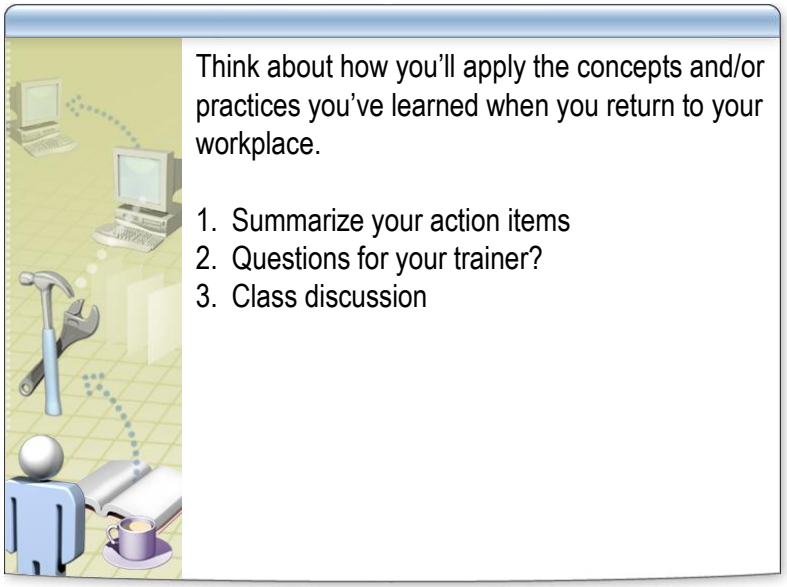


# Lab 1: Exercise 3




45

# Action Planning Exercise




When you return to your workplace, you'll want to use the concepts and practices you've learned to positively impact your IT environment. In this exercise you'll think about what you've learned and create action items that you can follow up on when you return to your workplace.

## Step 1: Summarize Your Action Items (5 Minutes)

How can you apply what you've learned to your workplace?

| | |
|---|---|
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |

### Step 2: Questions for your trainer? (5 Minutes)

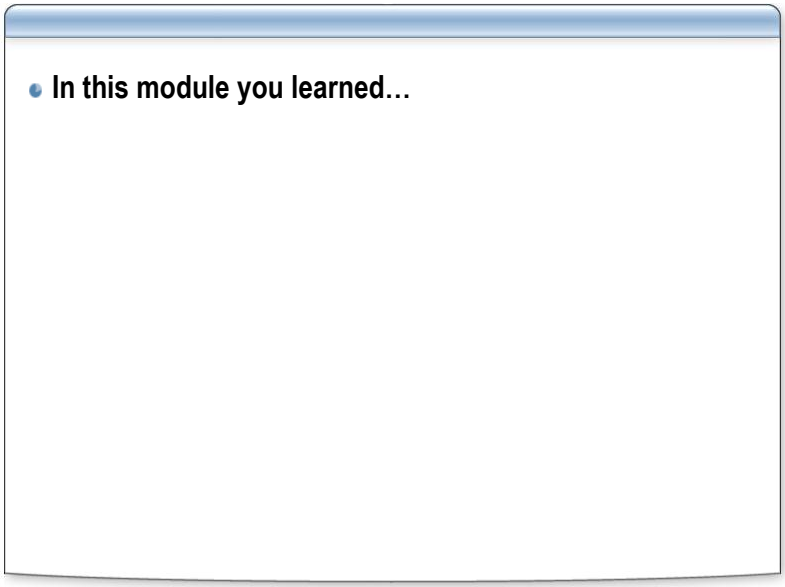
Record any questions you have about accomplishing the Action Items you listed above. The trainer will allow time for discussion before moving to the next module.

| | |
|---|---|
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |

### Step 3: Class Discussion

Notes:

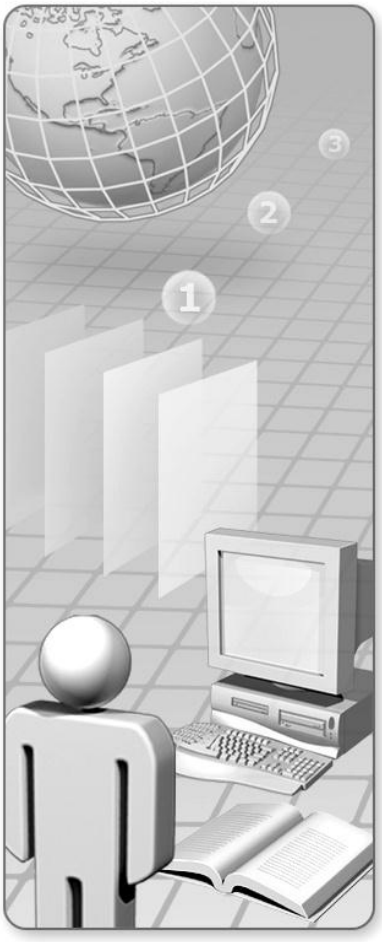
# Module Summary



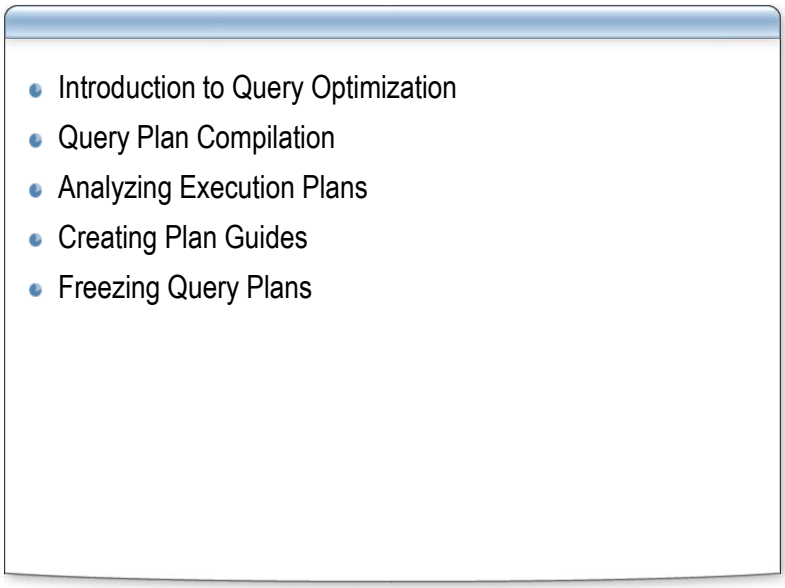

In this module we covered:

- The basic concepts of locking, including lock mode, lock resources, and lock compatibility.
- The impact of transaction isolation levels.
- Troubleshoot and resolve blocking and locking issues

# Module 5: Query Optimization



# Module Overview




## Introduction

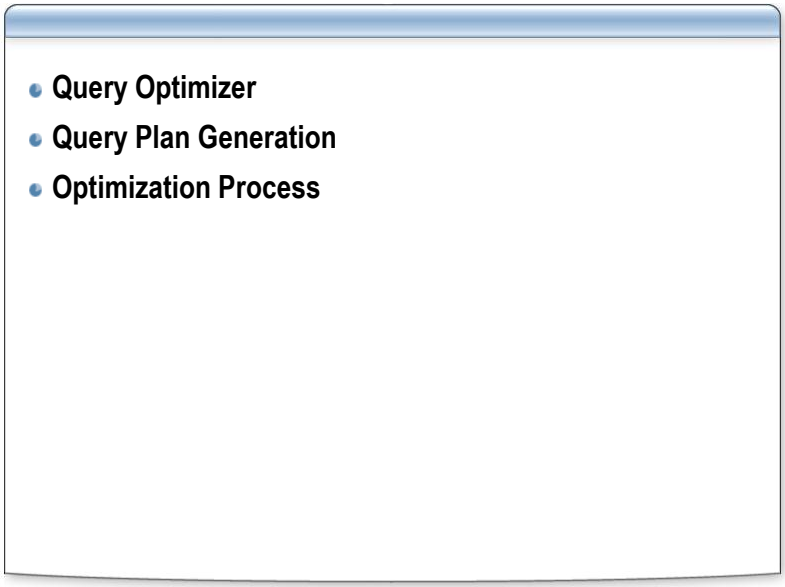
Getting the most from your database requires you to analyze your load, and optimize the database structure and queries for performance. This module introduces performance concerns and methods for optimizing your queries for performance.

## Objectives

After completing this module, you will be able to:

- Define the role of the query optimizer and how it works.
- Explain the importance of statistics and how they affect the choices made by the query optimizer.
- Analyze execution plans to identify potential problems.
- Optimize a query by creating a plan guide.

# Section 1: Introduction to Query Optimization




## Introduction

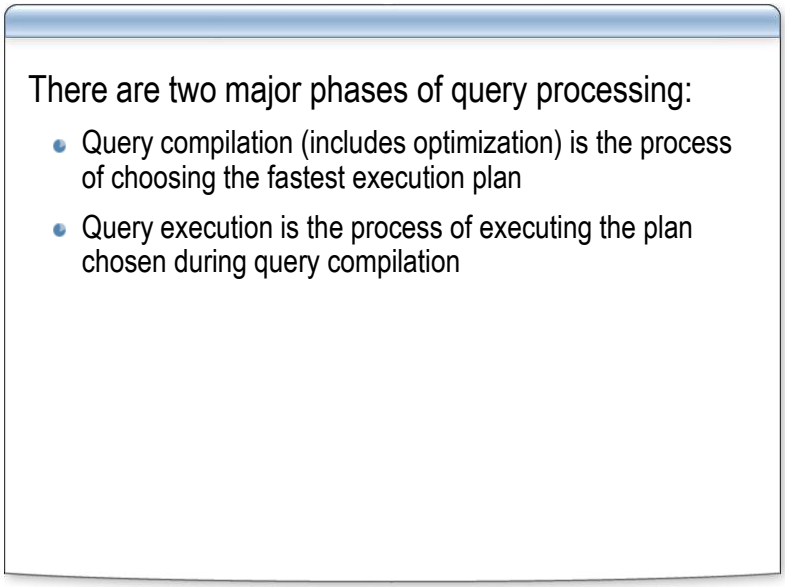
Once a query is submitted, SQL must develop a strategy to resolve it. Some queries can be quite complex. This section describes the process followed by SQL to develop an optimal strategy for returning the correct results.

## Objectives

After completing this section, you will be able to:

- Define the two major phases of query processing in SQL Server.
- Explain the high-level batch compilation and recompilation process that SQL Server performs when generating a query plan.
- Explain the main steps in the query optimization process.
- Explain the three phases in the full query optimization process.

# Query Optimizer




***Query optimization*** is the process of choosing the fastest execution plan. In the optimization phase, the query processor chooses:

- The indexes, if any, to be used.
- The order in which joins are executed.
- The order in which constraints, such as WHERE clauses, are applied.
- The algorithms that are most likely to lead to the best performance, based on costing information derived from statistics.

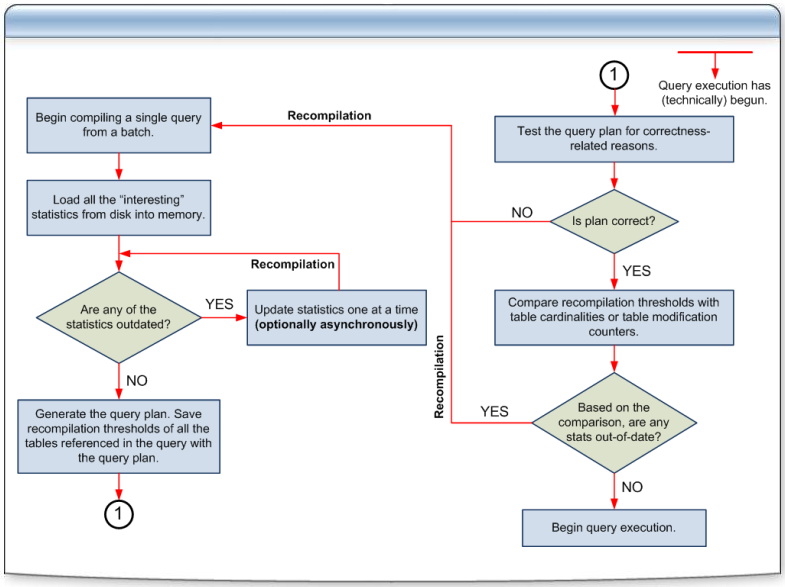
The SQL Server query optimizer is a cost-based optimizer, which means that it tries to come up with the cheapest execution plan for each SQL statement. The cost of the plan reflects the estimated processing cost to execute.

***Query execution*** is the process of executing the plan that is chosen during query optimization. The query execution component also determines the techniques that are available to the query optimizer.

Query compilation and execution are distinct phases of query processing. The gap between the time when SQL Server compiles a query and the time that the query is executed can be as short as a few microseconds (ms) or as long as several days. Therefore, during compilation (which includes optimization), SQL Server needs to determine what kind of available information will be useful during the compilation. Not everything that is true at compilation time will be true at execution time. For example, the query optimizer might take into account how many concurrent users are trying to access the data that your

query is trying to access, but the number of concurrent users can change dramatically from moment to moment. Compilation and execution are two separate activities, even when you are submitting an ad hoc SQL statement through SQL Server Management Studio and running it immediately.

# Query Plan Generation

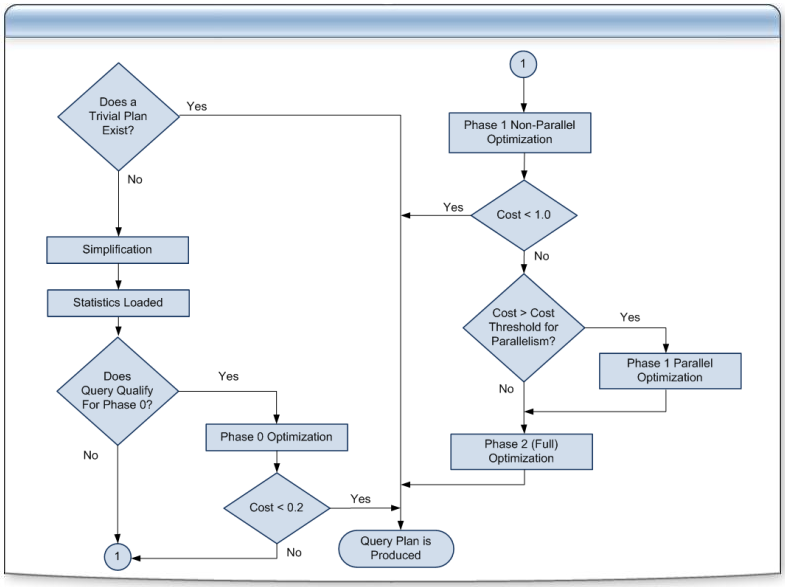



The following steps describe the batch compilation and recompilation process in SQL Server at a high level. The main processing steps (individual steps will be described in detail later on in this document) are as follows:

1. SQL Server begins compiling a query. (As previously described, a batch is a unit of compilation and caching, but individual statements in a batch are compiled one after the other.)
2. All the index or column statistics that may be used to generate an optimal query plan are loaded from disk into memory.
3. If any of the statistics are outdated, they are updated one at a time. The query compilation waits for the updates to finish. In SQL Server 2005 and later, statistics may, optionally, be updated asynchronously. That is, the query compilation thread is not blocked by statistics updating threads. The compilation thread proceeds with stale statistics.
4. The query plan is generated. Recompilation thresholds of all the tables referenced in the query are stored along with the query plan.
5. At this point, query execution has technically begun. The query plan is now tested for correctness-related reasons (for example, if the underlying table structure has changed).

6. If the plan is not correct, for any of the correctness-related reasons, a recompilation is started. Notice that, because query execution has technically begun, the compilation that has just been started is a recompilation.
7. If the plan is ***correct***, then various recompilation thresholds are compared with either table cardinalities or table modification counters (rowmodctr in SQL Server 2000 and colmodctr in SQL Server 2005).
8. If any of the statistics are determined to be out-of-date by the comparisons performed in Step 7, it results in a recompilation.
9. If all the comparisons in Step 7 succeed, actual query execution begins.

# Optimization Process




Each possible execution plan has an associated cost, in terms of the amount of computing resources used. The query optimizer must analyze the possible plans and choose the one with the lowest estimated cost. Some complex SELECT statements have thousands of possible execution plans. In these cases, the query optimizer does not analyze all possible combinations. Instead, it tries to find an execution plan that has a cost that is reasonably close to the theoretical minimum.

The lowest estimated cost is not simply the lowest resource cost; the query optimizer chooses the plan that most quickly returns results to the user with a reasonable resource cost. For example, processing a query in parallel by using multiple CPUs simultaneously for the same query typically uses more resources than processing it serially by using a single CPU, but the query completes much faster. So, the query optimizer chooses a parallel execution plan to return results, if the load on the server will not be adversely affected.

Optimization itself involves several steps. These are described below.

## Trivial optimization

The first step is called ***trivial plan optimization***. The idea behind trivial plan optimization is that cost-based optimization is expensive to run. The query optimizer can try many possible variations in looking for the cheapest plan. If SQL Server knows that there is only one really viable plan for a query, it can avoid a lot of work. A common example is a query that consists of an INSERT with a VALUES clause. There is only one possible plan. Another example is a SELECT, for which all the columns are among the keys of a

unique composite index, and that index is the only one that is relevant. No other index has that set of columns in it. In these two cases, SQL Server should just generate the plan and not try to find something better. The trivial plan query optimizer finds the really obvious plans that are typically very inexpensive. This saves the query optimizer from having to consider every possible plan, which can be costly and can outweigh any benefit provided by well-optimized queries.

### Simplification

If the trivial plan optimization phase does not find a simple plan, SQL Server can perform some simplifications. These simplifications are usually syntactic transformations of the query itself, to look for commutative properties and operations that can be rearranged. SQL Server can perform operations that do not require looking at the cost or analyzing what indexes are available, but that can result in a more efficient query. SQL Server then loads up the metadata, including the statistical information on the indexes. Then, the query optimizer goes through a series of phases of cost-based optimization.

### Cost-based optimization

The cost-based query optimizer follows a set of transformation rules that apply various permutations of data access strategies, join orders, aggregation placement, subquery transformations, and other rules that guarantee that a correct result is returned.

Because of the number of potential plans in SQL Server, the query optimizer cannot evaluate all possibilities in the process of producing a single plan. If it did, the optimization process could take much longer than the execution of the query. Therefore, optimization is broken up into three search phases.

Each search phase is a set of rules. After each phase, SQL Server evaluates the cost of the cheapest plan at that point, and if the plan is cheap enough, SQL Server executes that plan. If the plan is not cheap enough, the query optimizer runs the next phase which involves another set of usually more complex rules.

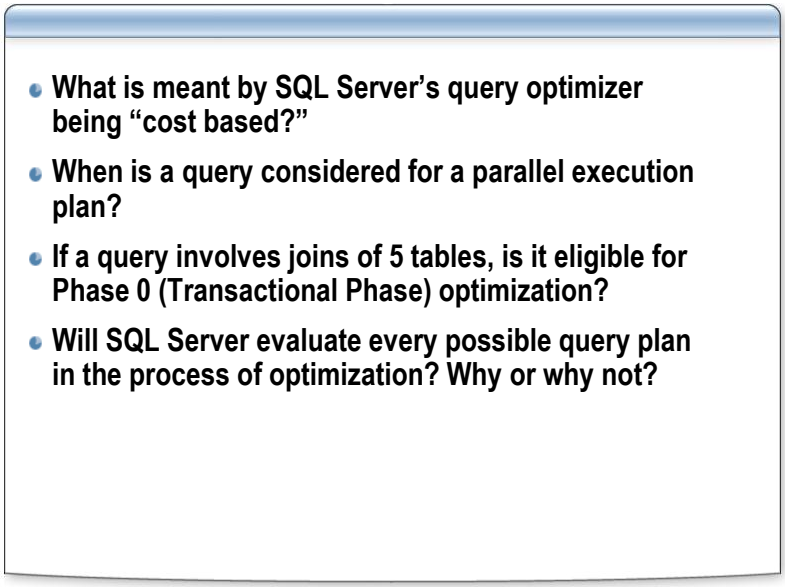
The three search phases are described below.

- **Phase 0 – Transactional Processing Phase**: Many queries, even though they are complicated, have very cheap plans. This first phase of cost-based optimization, Phase 0, contains a very limited set of rules and is applied only to queries with at least four tables. Join reordering alone generates many potential plan candidates; therefore, the query optimizer will evaluate only a limited number of join orders at this point, and it considers only hash and nested loop join strategies. If Phase 0 finds a plan with an estimated cost below 0.2, the optimization ends and the query is executed. Queries with final query plans produced by this phase are typically found in transaction processing applications; therefore, this phase is sometimes also referred to as the *Transaction Processing* phase.
- **Phase 1 – Quick Plan Optimization Phase**: During this phase, the query optimizer uses more transformation rules and evaluates different join orders. At the end of this phase, if the best plan costs less than 1.0, the optimization ends.

At this point in the optimization process, the query optimizer has considered only non-parallel execution plans. But if more than one CPU is available to SQL Server and the least expensive plan produced by Phase 1 non-parallel optimization costs more than the cost threshold for parallelism, Phase 1 is repeated with the goal of finding the best parallel plan.

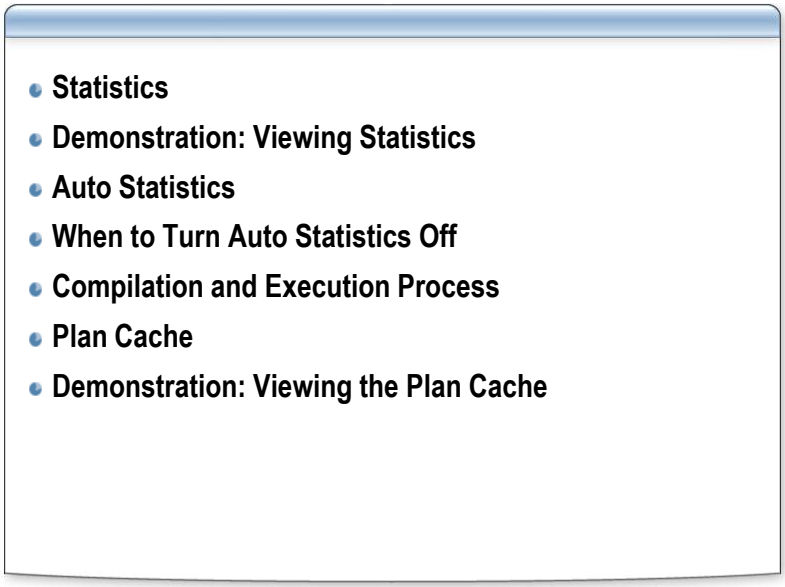
- **Phase 2 – Full Optimization Phase**: At the end of Phase 1 optimization, the costs of the serial and parallel plans produced are compared, and Phase 2 is executed for the cheaper plan. During Phase 2, outer join reordering and automatic indexed view substitution for multitable views are also considered.

# Section 1 Review




- What is meant by the SQL Server query optimizer being ***cost based***?
- When is a query considered for a parallel execution plan?
- If a query involves joins of five tables, is it eligible for Phase 0 (Transactional Phase) optimization? If yes, will SQL Server evaluate every possible query plan in the process of optimization? Why or why not?

# Section 2: Query Plan Compilation




## Introduction

In order to determine what indexes are useful, and what algorithm should be used to access and join data, SQL needs to be able to evaluate any existing indexes for usefulness. However, it would be extremely expensive for SQL to make a new evaluation of the data each time a query plan was submitted.

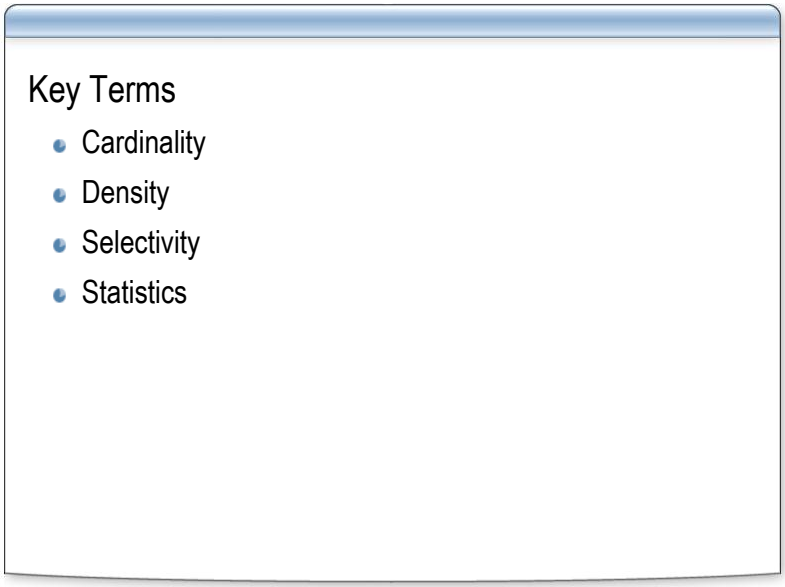
This section explains how SQL makes these evaluations, and how you can help in the process by keeping the metadata updated.

## Objectives

After completing this section, you will be able to:

- Explain the role of indexes and index statistics in compiling a query plan.
- View the statistics in an index.
- Keep statistical information up-to-date.
- Determine when to disable the Auto Statistics option.
- Explain the query plan compilation and execution process.
- Differentiate between query (or compiled plans) and execution contexts in an execution plan.
- View the execution plans stored in the plan cache by using the **sys.dm_exec_cached_plans** and **sys.dm_exec_query_stats** dynamic management views (DMVs).

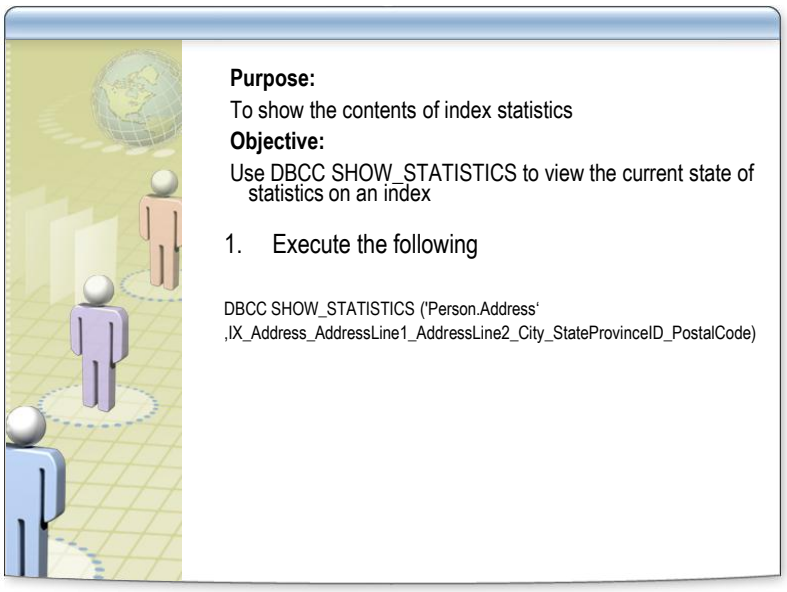
# Statistics




Some of the key terms relating to statistics include the following:

- **Cardinality**: Refers to the number of unique values within a data set.
- **Density**: Refers to the uniqueness of values within a data set. The density of an index is calculated by dividing the number of rows that correspond to a given key value by the number of rows in the table. For a unique index, this amounts to dividing 1 by the total row count of the table. Density values range from 0 through 1; lower densities are better.
- **Selectivity**: Is a measure of the number of rows that will be returned by a particular query criterion. The term selectivity expresses a relationship between query criteria and the key values in an index. It is computed by dividing the number of keys being requested by the number of rows that they access. Query criteria (usually specified in a WHERE clause) that are highly selective are the most useful to the query optimizer, because they enable it to predict, with certainty, how much I/O is required to satisfy a query.
- **Statistics**: Are a histogram, consisting of an even sampling of values for the index key (or the first column of the key for a composite index), based on the data that was current at the time of building statistics. To fully estimate the usefulness of an index, the query optimizer also needs to know the number of pages in the table or index.

# Demonstration 1: DBCC SHOW_STATISTICS




**DBCC SHOW_STATISTICS**: Can be used to determine when statistics were last updated, how many rows were sampled to build the statistics, and what steps were built.

> **Note**: For more information on this command, refer to the topic *DBCC SHOW_STATISTICS* in SQL Server Books Online.

After the query optimizer finds an index that is potentially useful in resolving the data, it evaluates the index, based on the selectivity of the clause. The query optimizer checks the index statistics—the histogram of values accessible through the index rows in the sysindexes table. The histogram is created when the index is created on existing data, and the values are refreshed each time UPDATE STATISTICS runs. If the index is created before data exists in the table, no statistics appear. The statistics will be misleading if they were generated when the dispersion of data values was significantly different from what appears in the current data in the table. However, SQL Server detects whether statistics are not up-to-date, and, by default, it automatically updates them during query optimization.

In a query window, run the following:

```
USE AdventureWorksPTO
go


DBCC SHOW_STATISTICS ('Person.Address'
```

```
        , IX_Address_AddressLine1_AddressLine2_City_StateProvinceID_PostalCode)
```

When you run the command shown above, you will get the following three resultsets:

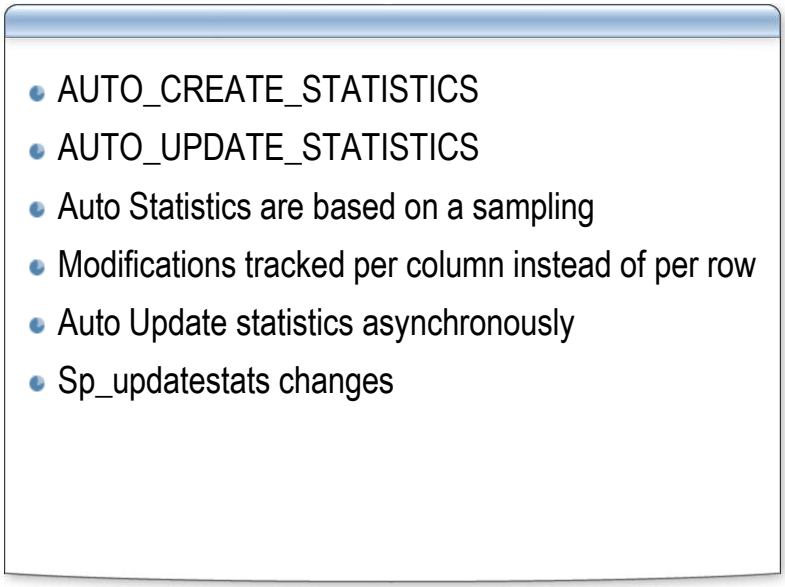
1. **STAT_HEADER**: In this resultset, you can see:
   - The names of the statistics
   - The last time the statistics were updated
   - The number of rows in the index the last time the statistics were updated
   - The number of row samples during the update
   - The number of steps in the histogram (explained below)
   - The density of the first column excluding the equivalent rows
   - The average key length
   - Whether or not the first index key is a string value
2. **DENSITY_VECTOR**: This resultset contains as many rows as there are keys in the index. For each combination of keys in this resultset, there is an ALL_DENSITY, which differs from the density in the STAT_HEADER because the former includes the equivalent rows.
3. **Histogram**: Note that the histogram is built only on the first column in the index. SQL will need to create statistics on subsequent columns separately. There is one row for each step in the histogram. Note that this matches the ***steps*** value in the STAT_HEADER resultset. RANGE_HI_KEY is the highest value in the histogram step. RANGE_ROWS is the number of rows estimated in this range that is not equal to RANGE_HI_KEY. EQ_ROWS is the number of rows estimated in this range that is equal to RANGE_HI_KEY.

This statistical information is what SQL will use to estimate the number of rows returned in a query.

For example, when you run the following query, SQL looks at the histogram to estimate the number of rows where AddressLine1 is **1039 Adelaide St.**:

```
Select * from Person.address where AddressLine1 = '1039 Adelaide St.'
```

# Auto Statistics




## AUTO_CREATE_STATISTICS

When this option is set to ON (the default), the SQL Server query optimizer automatically creates statistics on columns referenced in the WHERE clause of a query. As you would recall from the demonstration in the previous topic, a histogram is built only on the first column in a composite index. If the WHERE clause in your query references the first and second column of that index, if AUTO_CREATE_STATISTICS is turned on, and if no other index exists on the second column, SQL will create statistics on the second column so that it can better estimate the number of rows returned by the combination of the first and second columns.

Adding statistics improves query performance because the SQL Server query optimizer can better determine how to evaluate a query. Statistics are now created in the ***sys.stats*** view. In earlier editions, statistics were created in sysindexes. However, the naming convention still applies. All system created statistics start with **_WA_**.

## AUTO_UPDATE_STATISTICS

Whenever a new index is built, statistics on indexes are created automatically. It is also possible to create and maintain statistics on other columns.

When the AUTO_UPDATE_STATISTICS option is set to ON (the default), existing statistics are automatically updated if the data in the index has changed significantly. SQL Server keeps a counter of the modifications that have been made to a table and uses it to determine when statistics are outdated. When this option is set to OFF, existing statistics are not automatically updated.

The query optimizer can be extremely sensitive to statistical information that is provided on tables and indexes. Without correct and up-to-date statistical information, SQL Server query optimizer may not be able to determine the best execution plan for a particular query.

To keep statistical information as up-to-date as possible, SQL Server uses AutoStats. Autostats uses the SQL Server monitoring of table column modifications to automatically update the statistics for a table when a certain change threshold is reached.

> **Note**: It is better not to use sysindexes.rowmodctr to determine when Autostats fires.

Statistical information is updated when approximately 20 percent of the data rows have changed. When statistics are updated, a random sampling, across data pages, is taken either from the table or from the smallest non-clustered index on the columns needed by the statistics. Tables that are smaller than 8 megabytes (MB) are always fully scanned to gather statistics.

You can also manually update statistics by using UPDATE STATISTICS. For large tables, a random sampling may not produce accurate statistics. Therefore, for large tables, you may need to use the SAMPLE or FULLSCAN option on UPDATE STATISTICS to specify a sample size higher than the default.

## AUTO_UPDATE_STATISTICS_ASYNC

SQL Server 2005 introduces AUTO_UPDATE_STATISTICS_ASYNC, which requires you to enable AUTO_UPDATE_STATISTICS. You can set both these options ON by using the ALTER DATABASE command as follows:

```
alter database MyDB set AUTO_UPDATE_STATISTICS ON,
        AUTO_UPDATE_STATISTICS_ASYNC ON
```

Before running a query, SQL Server checks the plan to determine if any of the referenced objects exceed the threshold of stale statistics. If the threshold is exceeded, the server lines up the job in the background job queue to rebuild the statistics, but continues with compilation, without waiting for the background job to complete.

> **Important**: If the statistics for an object which has been the target of a Data Definition Language (DDL) operation during the current transaction are stale, then the statistics are still built synchronously. This prevents situations where the table might have been altered to add a column, and the change would not yet be visible to a background job running under a separate transaction. However, the most common scenario in which this might apply is statistics updates for temp tables, created and populated in the same transaction (for example, a stored procedure). For a simple case such as a temp table, an asynchronous statistics update would be too late to be useful.

You can view the currently queued jobs by using the sys.dm_exec_background_job_queue dynamic management view (DMV), which is currently used only for asynchronous update statistics jobs. The database_id column displays which database the job will run against, the object_id1 column displays the object ID of the table or view, and the object_id2 column displays the statistics ID that needs to be updated.

The query below lists all asynchronous update statistics:

```
select db_name(database_id) as dbname, object_name(object_id1) table_view_name,
object_name(object_id2) as stats_name
from sys.dm_exec_background_job_queue
```

The **auto stats** trace events that are fired on asynchronous updates of statistics do not currently populate the TextData column to indicate which statistics or columns are being updated. You should instead use the DBID, ObjectID, and IndexID columns of the autostats trace event and look this information up in the **sys.stats** system view.

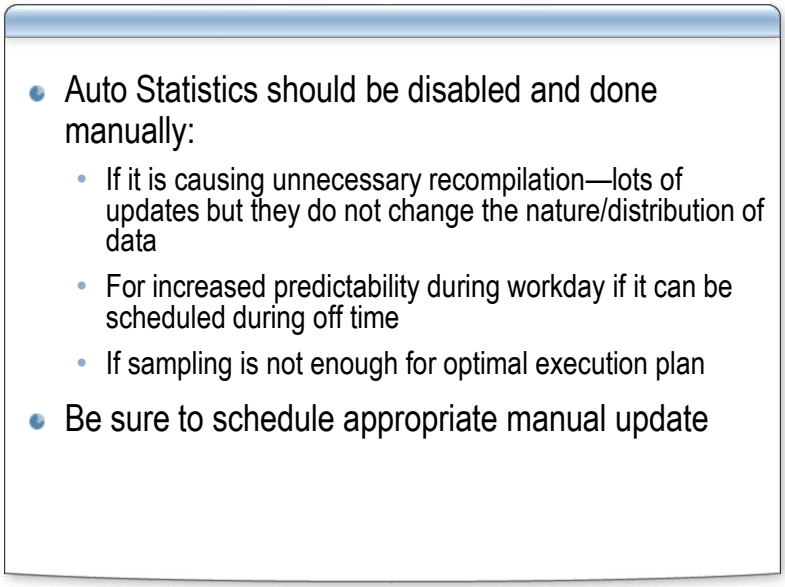
SQL Server does not have any type of priority system available; therefore, the background jobs end up running at the same priority as other sessions. The stats job sets its deadlock priority, so that it would be chosen as the preferred deadlock victim in the event of a deadlock.

## SP_UPDATESTATS

In SQL Server 2000, sp_updatestats would update the statistics for all the objects in a database, regardless of whether there had been any changes to the table (that is, rowmodctr was zero). However, in SQL Server 2005, if the sysindexes.rowmodctr value is zero, then the index or statistics is skipped because its statistics are already fully up-to-date.

If you have an index or statistic that you do not want SQL to update via AUTO_UPDATE_STATISTICS, you can specify the NORECOMPUTE flag, but you must specify it for each UPDATE STATISTICS operation that you run. This flag is useful when you encounter a situation where the sampled statistics for a given index cause a problem, but FULLSCAN statistics work well. By setting this flag, you could prevent the system from automatically updating statistics and reverting to sample statistics. Because sp_updatestats is just a stored procedure that issues dynamic UPDATE STATISTICS statements, it must also specify the flag to preserve a NORECOMPUTE option. The SQL Server 2000 sp_updatestats did not preserve NORECOMPUTE, but the SQL Server 2005 and SQL Server 2008 sp_updatestats does, when the database is not in 80 compatibility mode.

# When to Turn Auto Statistics Off




To provide the up-to-date statistics, the SQL Server query optimizer must make smart query optimization decisions. Therefore, it is generally best to leave the AUTO_UPDATE_STATISTICS database option ON (the default setting). This helps ensure that the query optimizer statistics are valid and that queries are properly optimized when they are run.

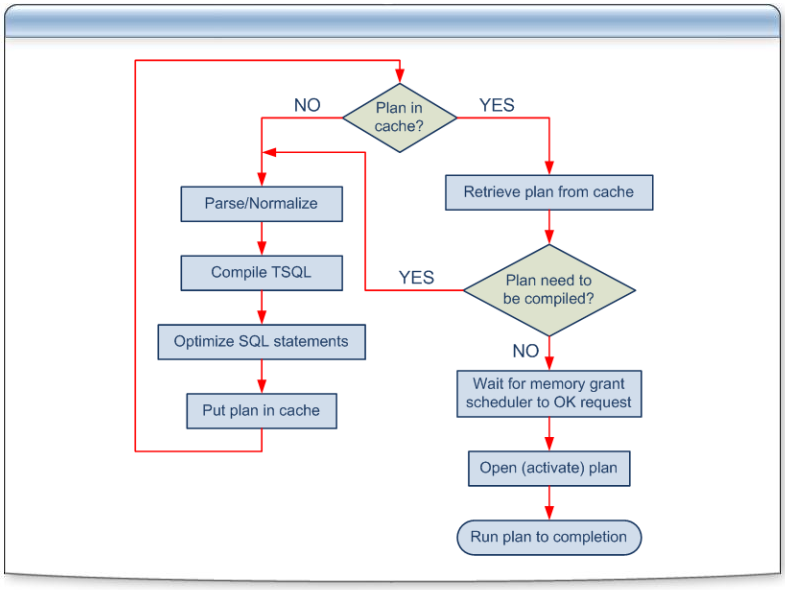
There are some scenarios, however, where it might be appropriate to turn AUTO_UPDATE_STATISTICS OFF.

When a SQL Server database is under very heavy load, sometimes the AUTO_UPDATE_STATISTICS can update the statistics on large tables at inappropriate times. If you find that the auto update statistics feature is running at inappropriate times, you might want to turn it OFF, and then manually update the statistics (by using UPDATE STATISTICS). But before disabling this feature, you should consider scheduling manual updates of statistics during less busy times, and leaving AUTO_UPDATE_STATISTICS enabled just as a safety valve. If statistics are updated manually when the load is lower, there is a lesser chance that the modification threshold that triggers an automatic update of statistics will be crossed during heavy processing times. In this case, AUTO_UPDATE_STATISTICS can still prevent processing on stale statistics, if necessary, but the extra load of updating statistics during the heaviest processing may no longer be an issue. To determine whether auto statistics is enabled for a table, you should use **sp_autotstats**.

As with many other optimization issues, you must test to see whether turning this option ON or turning it OFF is more effective for your environment.

> **Note**: If you do turn AUTO_UPDATE_STATISTICS off, be sure to implement an appropriate manual update schedule.

# Compilation and Execution Process




SQL Server is running, statistics are up-to-date, queries have been optimized and cached. Your application submits a new query to SQL Server. SQL Server receives this query, and begins processing it. If an execution plan for this query is not in cache, SQL Server must create one. After the plan has been put in cache, SQL Server determines whether anything has changed and whether the plan must be recompiled before executing it. Even though there might be only microseconds separating compilation from execution, a user may run a DDL statement that adds an index to a crucial table. While unlikely, SQL Server must account for this possibility.

## Recompiling a stored plan

There are a few events that cause SQL Server to recompile a stored plan:

- **Metadata changes**, such as adding or dropping indexes, is the most likely reason. You need to make sure that the plan being used reflects the current state of the indexes.
- **Statistical changes** also cause recompilation. SQL Server keeps quite a bit of histogram information on the data that it processes. If the data distribution changes a lot, you may need a different query plan to get the most efficient execution. If SQL Server has updated the statistics on an index used in the plan, a subsequent use of that plan invokes recompilation, based on the newly distribution information.

A change of actual parameter will not cause a plan to be recompiled. In addition, environmental changes such as the amount of memory available or the amount of needed data that is already in cache will not cause recompilation.

## Running large queries

Execution is straightforward, and if queries are very simple queries, such as ***insert one row*** or a singleton select from a table with a unique index, the processing is very simple. However, many queries require, or at least could benefit from, large amounts of memory to run efficiently.

Starting with SQL Server 7.0, the SQL Server developers added the ability to use large amounts of memory for individual queries. However, this led to another problem. When you allow queries to use very large amounts of memory, you need to decide how to arbitrate this memory use among all queries that might need that memory. SQL Server handles this as follows:

1. When a query plan is optimized, the query optimizer determines two pieces of information about memory use for that query:
   - It picks the minimum amount of memory that the query needs in order to run effectively, and stores this value with the query plan.
   - It determines the maximum amount of memory that the query could benefit from. For example, giving a sort operation 2 gigabyte (GB) of memory does not offer any benefit, if the entire table being sorted fits into 100 MB. You really just want the 100 MB, so that should be the value for the maximum amount of useful memory that is stored with the query plan.
2. When SQL Server is ready to run the plan, the plan gets passed to a routine called the ***memory grant scheduler***. If the query does not have a sort or a hash operation as part of the plan, SQL Server knows that it does not require large amounts of memory. In this case, the plan does not wait in the memory grant scheduler and can be run immediately. A typical transaction processing request bypasses this mechanism completely.

## Considerations for running large queries

If there are very large queries, you should run a few of them at a time and let them use more of the memory that they need. If at least half of the maximum memory requirement of the query is available, the query will run immediately with whatever memory it can get. Otherwise, the query will wait in the *memory grant queue* until half of the maximum memory requirement is available. At the same time, SQL Server will never allocate more than half of the remaining query memory to any one query. If you encounter many memory timeout errors (error 8645), this indicates that your queries were waiting too long in the memory grant scheduler.

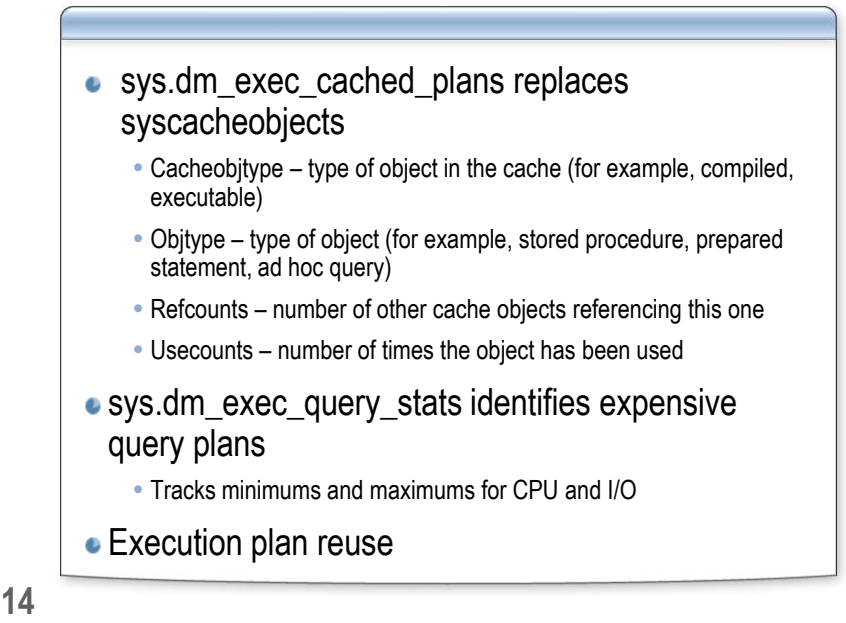
By default, a query will wait to run only for that amount of time that is proportional to its estimated cost; currently, that value is about 25 times the estimated number of seconds. With the default value, no query will wait less than 25 seconds because the query cost is rounded up to one if it is less than one. If the time limit is exceeded, error 8645 is generated.

## Running the plan

After the scheduler allocates memory for the execution of a query, SQL Server performs a step called *opening the plan*, which starts the actual execution. There onwards, the plan runs to completion. If your query is using the default ***result set model***, the plan will actually just run until it has produced all of its rows and these have all been sent back to the client.

However, if you are using the ***cursor model***, the processing is a bit different. Each client requests only one block of rows, not all of them. After each block is sent back to the client, SQL Server must wait for the client to request the next block. While it is waiting, the entire plan is quiesced, which means that some of the locks are released, some resources are given up, and some positioning information is cached. When the next block of rows is requested, this cached information enables SQL Server to resume where it left off.

# Plan Cache




SQL Server execution plans are broken into two parts—query or compiled plans, and execution contexts. Query or compiled plans are sharable by many users, whereas execution contexts are not. Query plans are stored in **sys.dm_exec_cached_plans**.

### Query or compiled plan

A query or compiled plan is a re-entrant, read-only data structure used by any number of users. It contains the data access strategy for the query. A query plan does not store any user context. There are never more than two copies of a query plan in the memory. There is possibly one copy for all serial executions and possibly one copy for all parallel executions. The parallel copy covers all parallel executions, regardless of their degree of parallelism.

### Execution context

The execution context data structures are reusable but not re-entrant. Consequently, when multiple users run a stored procedure concurrently, each user is given a separate execution context. So, there can be multiple execution contexts, especially in high concurrency environments. If a new user runs a stored procedure, and one of the existing execution contexts is not in use, this execution context is re-initialized with the context for the new user.

### sys.dm_exec_cached_plans

The sys.dm_exec_cached_plans DMV returns information about execution plans that are in cache. This DMV is similar to syscacheobjects in SQL Server 2000. One notable difference, however, from syscacheobjects is that the sys.dm_exec_cached_plans DMV

never displays execution contexts. An execution context is no longer a ***first class*** cache object in SQL Server 2005. The execution context maintains state information, such as the values of local variables and parameters for that execution plan, which is specific to a given invocation of a query. Given a compiled plan, an execution context is fairly inexpensive to create (by allocating some memory and initializing data structures).

To improve plan stability, SQL Server retains the compiled plans in cache as long as possible. The architecture was changed beginning with SQL 2005 so that compiled plans can keep pointers to the application execution contexts for that plan. Under memory pressure, the execution contexts are removed first because they can be easily regenerated. The compiled plans are only removed when there is still insufficient memory after removing the execution contexts.

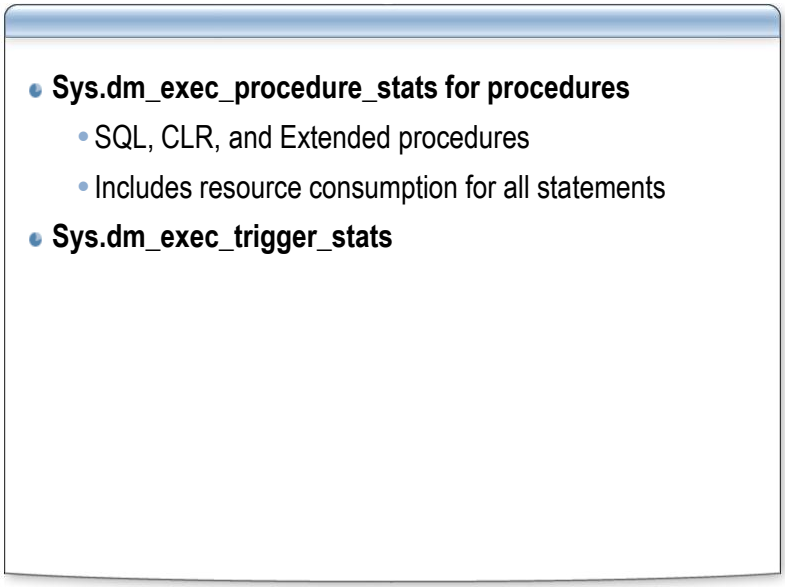
You can use the following query to find query plans that may run in parallel:

```
select
    p.*,
    q.*,
    cp.plan_handle
from
    sys.dm_exec_cached_plans cp
    cross apply sys.dm_exec_query_plan(cp.plan_handle) p
    cross apply sys.dm_exec_sql_text(cp.plan_handle) as q
where
    cp.cacheobjtype = 'Compiled Plan' and
    p.query_plan.value('declare namespace
p="http://schemas.microsoft.com/sqlserver/2004/07/showplan";
        max(//p:RelOp/@Parallel)', 'float') > 0
```

## sys.dm_exec_query_stats

The **sys.dm_exec_query_stats** DMV tracks statistics for cached query plans. You can use it to identify those query plans in the cache that consume the most CPU, time to run, and I/O. It tracks the CPU time in the ***worker_time** columns and breaks down I/O into logical and physical reads and writes. The sys.dm_exec_query_stats DMV tracks minimum and maximum usage for both categories.

# Identifying Expensive Procedures and Triggers




SQL Server 2008 has added 2 dmvs to identify expensive procedures and triggers.

## Expensive Procedures

To identify expensive procedures use sys.dm_exec_procedure_stats. This dmv can be used to identify expensive TSQL stored procedures, CLR stored procedures, or Extended stored procedures. The type and type_desc columns indicate which type of procedure it is.

This dmv is similar to sys.dm_exec_query_stats. However sys.dm_exec_procedure_stats includes consumption for all statements within a procedure not just the select, insert, update, and delete statements. If you issue a procedure which has multiple TSQL statements most of the statements in the batch will have the same plan_handle value in sys.dm_exec_query_stats. So to identify how expensive a procedure is you could use sys.dm_exec_query_stats and group by the plan_handle. However, this approach will only show resources consumed by select, insert, delete, and update statements.

Demo:

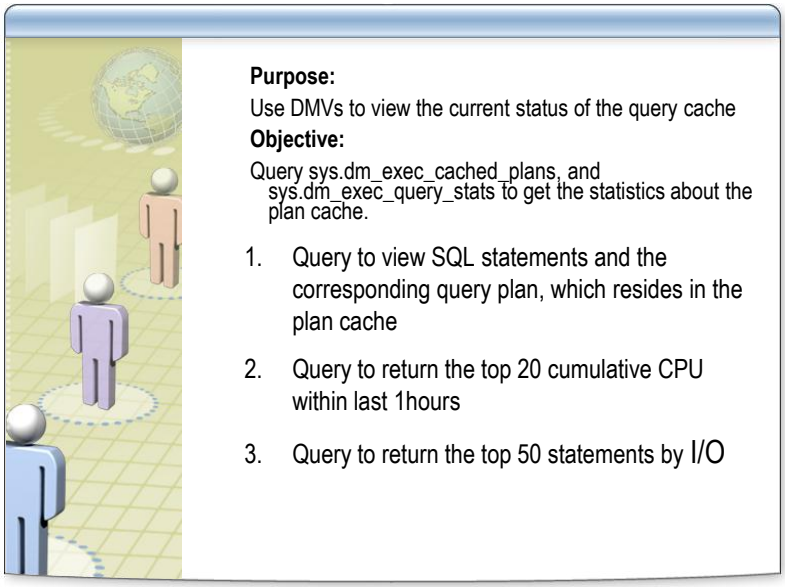
```
Use adventureworks
go
create proc mytestproc as
select * into mycontact
from Person.Contact
```

```
select lastname, firstname, emailaddress
from myContact
where LastName in ('smith','wesson')
create index anotherindex on mycontact (lastname, firstname, emailaddress)
go
dbcc freeproccache
go
exec mytestproc
-- note the total io and cpu time returned
select plan_handle, text,
SUM(total_logical_writes+total_logical_reads+total_physical_reads) as  [total io],
SUM(total_worker_time) as [total cpu]
from sys.dm_exec_query_stats
cross apply sys.dm_exec_sql_text(sql_handle)
group by plan_handle, text
order by [total io]
-- now use sys.dm_exec_procedure_stats and compare the total  io and cpu time
-- notice sys.dm_exec_procedure_stats includes the cpu and io taken to create the index; sys.dm_exec_query_stats doesn't
select plan_handle, (total_logical_writes+total_logical_reads+total_physical_reads) as [total io],
total_worker_time, text
from sys.dm_exec_procedure_stats
cross apply sys.dm_exec_sql_text(sql_handle)
```

## Expensive Triggers

To identify expensive triggers use sys.dm_exec_trigger_stats. This DMV is functionally equivalent to the new sys.dm_exec_procedure_stats but it provides the data about the overall time taken inside of triggers only. This information is available for both SQL and CLR triggers.

# Demonstration 2: Viewing the Plan Cache




To view SQL statements and the corresponding query plan, which resides in the plan cache, use the following queries:

```
select cacheobjtype, objtype, refcounts, usecounts, text, query_plan
from sys.dm_exec_cached_plans a
join  sys.dm_exec_query_stats b on a.plan_handle=b.plan_handle
CROSS APPLY sys.dm_exec_sql_text(b.sql_handle)
CROSS APPLY sys.dm_exec_query_plan(b.plan_handle)

--The query below returns the  Top 20 Cumulative CPU within last 1hours
SELECT last_execution_time, total_worker_time AS [Total CPU Time], execution_count,
total_worker_time/execution_count AS [Avg CPU Time],
text, qp.query_plan
FROM sys.dm_exec_query_stats AS qs
CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) AS st
CROSS APPLY sys.dm_exec_query_plan(qs.plan_handle) as qp
WHERE  DATEDIFF(hour, last_execution_time, getdate()) < 1   -- change hour time
frame
ORDER BY total_worker_time DESC;

--Example Top 50 statements by I/O
SELECT TOP 50
        (qs.total_logical_reads + qs.total_logical_writes) /qs.execution_count as
[Avg IO],
       substring (qt.text,qs.statement_start_offset/2,
              (case when qs.statement_end_offset = -1
              then len(convert(nvarchar(max), qt.text)) * 2
              else qs.statement_end_offset end -   qs.statement_start_offset)/2)
              as query_text,
       qt.dbid,
```

```
       qt.objectid
FROM sys.dm_exec_query_stats qs
cross apply sys.dm_exec_sql_text (qs.sql_handle) as qt
ORDER BY [Avg IO] DESC
```

## Reusing execution plans

In high concurrency simultaneous execution scenarios with plan reuse, there will be multiple execution plans for the same obj_id with usecounts greater than 1, thereby indicating plan reuse. Low usecount values might indicate plans with poor plan reuse.

The following query displays plans with a usecount less than 15:

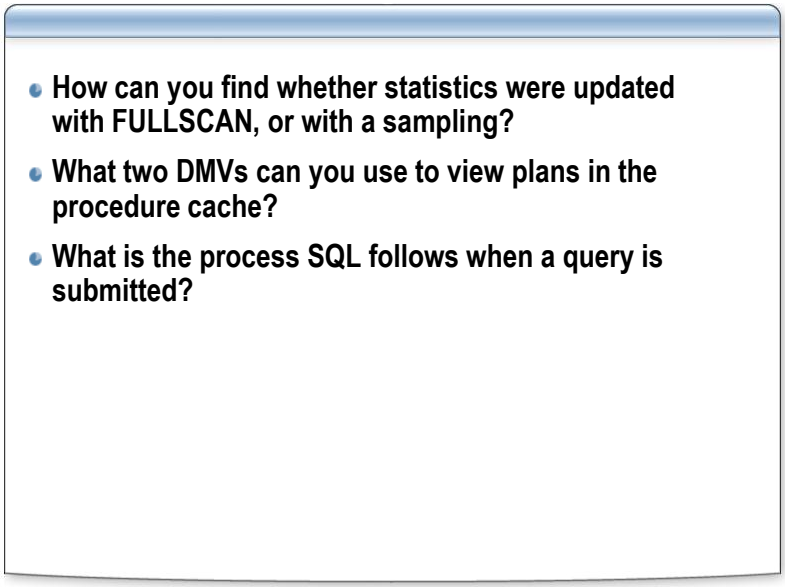
```
select cacheobjtype, objtype, refcounts, usecounts, text, query_plan
from sys.dm_exec_cached_plans a
join  sys.dm_exec_query_stats b on a.plan_handle=b.plan_handle
CROSS APPLY sys.dm_exec_sql_text(b.sql_handle)
CROSS APPLY sys.dm_exec_query_plan(b.plan_handle)
where usecounts <15
```

SQL Server plan reuse is determined by the text matching of SQL strings. If you see multiple compile plans with very similar SQL strings, this can indicate a plan reuse problem.

To determine how the plan cache is distributed, you need to identify how many rows are in the cache for each of the following object types:

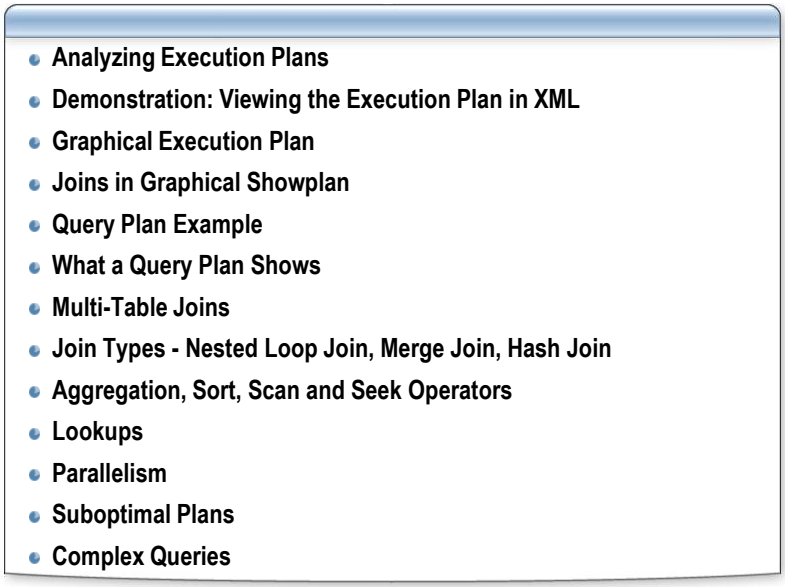
- Ad hoc
- Prepared
- Stored procedures

## Section 2 Review




- How can you determine whether statistics were updated with FULLSCAN or with sampling?
- What two DMVs can you use to view plans in the procedure cache?
- What is the process followed by SQL when a query is submitted?

# Section 3: Analyzing Execution Plans




## Introduction

Understanding execution plans is central to your ability to optimize SQL Server performance. In this section you will learn to read and understand query execution plans and work with SQL to optimize data access and retrieval.

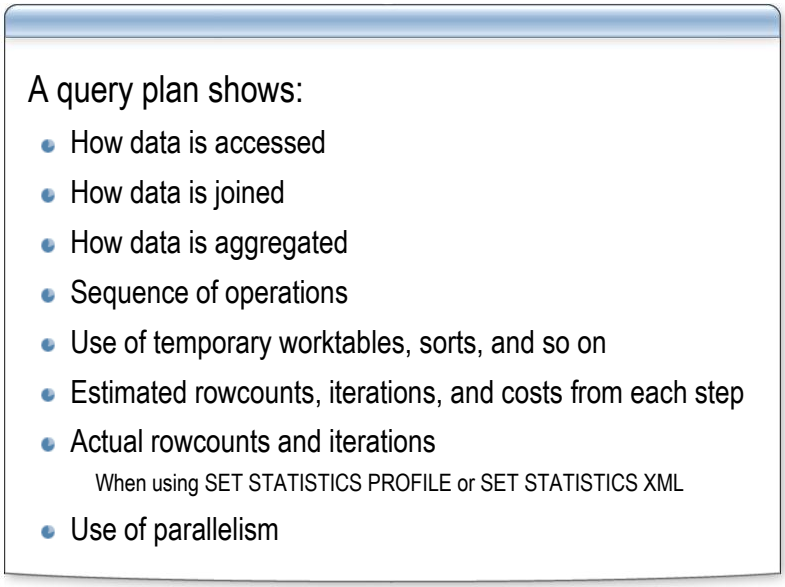
## Objectives

After completing this section, you will be able to:

- Analyze execution plans by reviewing the query processing output returned by the STATISTICS IO option.
- View an execution plan in XML.
- Display an execution plan graphically
- Differentiate between an estimated query plan and an actual query plan.
- Analyze a graphical execution plan.
- Identify the contents of a query plan.
- Explain how the query processor handles multiple-table joins.
- Compare the different join algorithms in SQL Server.
- Explain the characteristics of nested loop, merge, and hash joins.
- Describe the primary task of the aggregation operator.
- Describe the purpose of the sort operator.
- Differentiate between scans and seeks.

- Explain when scans and seeks are helpful and when lookups or scans should be eliminated.
- Identify the various options available in a suboptimal execution plan to create a more efficient plan.
- Explain why the query optimizer may choose a suboptimal query plan.
- List the various considerations for optimizing complex queries.

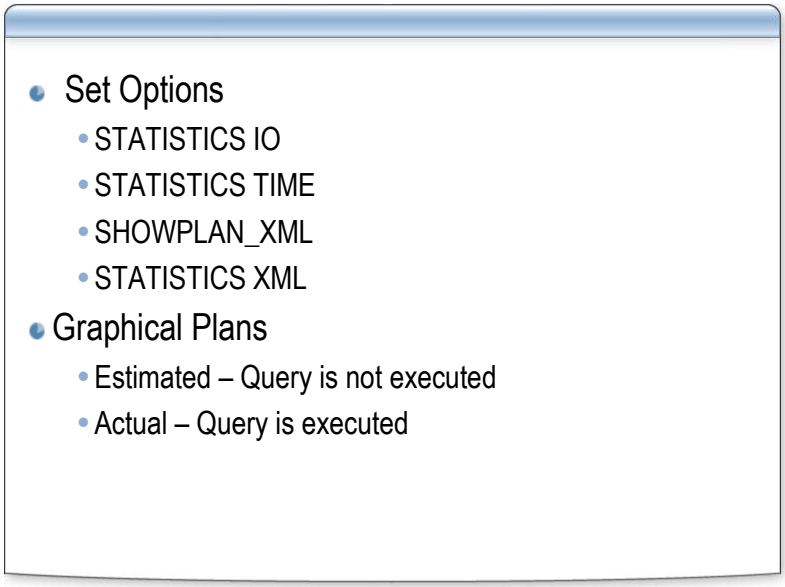
# Contents of a Query Plan




A query plans shows:

- How data is accessed, including where indexes are used and whether the query performs a seek or a scan.
- How data is joined, including the order in which the tables joined and the algorithm used to join the tables.
- How data is aggregated, including the aggregation algorithm used, if the query contains an aggregate.
- The sequence of operations.
- The use of temporary work tables. For example, the query optimizer may create temporary work tables if the query contains a sort clause.
- Estimated rowcounts, including how many rows the query optimizer estimated each operation within the query plan would return.
- Actual rowcounts, including the number of rows actually returned by each operation within the query plan.
- Whether or not a parallel query plan was used.

# Analyzing Execution Plans




Before you add an index or denormalize to make a query faster, you should understand how a query is currently being processed. You should also get some baseline performance measurements so that you can compare the behavior both before and after making your changes. SQL Server provides a few **SET** options for monitoring queries. The most useful ones include:

- STATISTICS IO
- STATISTICS TIME
- SHOWPLAN_XML
- STATISTICS XML

If you enable any of the ***SET STATISTICS*** options before you run a query, you will receive additional output. If you enable any of the ***SET SHOWPLAN*** options, the query will not be run, but instead, the information about the query will be displayed. Typically, you run your query with these options set in a tool, such as the Query window in SQL Server Management Studio. When you are satisfied with your query, you can cut and paste it into your application or into the script file that helps you create stored procedures. If you use the SET commands to turn these options ON, they apply only to the current connection.

## STATISTICS IO

STATISTICS IO is not related to the statistics used for storing histograms and density information in **sys.stats**. This option provides statistics on the amount of I/O work SQL Server performed to process a query. When this option is set to ON, there is a separate

line of output for each query in a batch that accesses any data objects. (No output is returned for statements that do not access data, such as PRINT, SELECT the value of a variable, or a call to a system function that does not access data.) The output from SET STATISTICS IO ON includes the Logical Reads, Physical Reads, Read Ahead Reads, Scan Count, lob logical reads, lob physical reads, and lob read-ahead reads values.

- Logical Reads

  The Logical Reads value indicates the total number of pages accessed to process the query. Every page is read from the data cache, whether or not it was necessary to bring that page from disk into the cache for any given read. This value is always at least as large as, and is usually larger than, the value for Physical Reads. The same page can be read many times (such as when an index is used to perform a key lookup for each row returned in a query). Therefore, the count of Logical Reads for a table can be greater than the number of pages in that table.

- Physical Reads

  The Physical Reads value indicates the number of pages that were read from disk; it is always less than or equal to the value of Logical Reads. Remember that the value for Physical Reads can vary greatly and decreases substantially with the second and subsequent execution, because the data is brought into the buffer by the first execution. The value is also affected by other SQL Server activities and can appear low if the page was preloaded by *read ahead* activity. So, you probably will not find it useful to do a lot of analysis of physical I/O on a per-query basis.

  When you are looking at individual queries, the Logical Reads value is usually more interesting, because the information is consistent. Physical I/O and achieving a good cache-hit ratio is crucial, but they are more interesting at the server level. You should pay close attention to Logical Reads for each important query, and pay close attention to physical I/O and the cache-hit ratio for the server as a whole.

- Read Ahead Reads

  The Read Ahead Reads value indicates the number of pages that were read into the cache by using the read ahead mechanism, while the query was being processed. These pages are not necessarily used by the query. If a page is ultimately needed, a Logical Reads is counted, not a Physical Reads. A high value indicates that the value for Physical Reads is probably lower, and the cache-hit ratio is probably higher, than if a read ahead was not done. In this situation, you should not infer from a high cache-hit ratio that your system will not benefit from additional memory. The high ratio might be driven by the read ahead mechanism bringing much of the needed data into cache. That is a good thing, but it might be better if the data simply remains in cache from previous use. You might achieve either the same or a higher cache-hit ratio without requiring the Read Ahead Reads.

  You can think of read ahead as simply an optimistic form of physical I/O. In full or partial table scans, the Index Allocation Maps (IAMs) of the table are consulted to

determine which extents belong to the object. The extents are read with a single 64 kilobytes (KB) scatter read, and because of the way that the IAMs are organized, they are read in disk order. If the table is spread across multiple files in a file group, the read ahead attempts to keep at least eight of the files busy, instead of sequentially processing these files. Read Ahead Reads are asynchronously requested by the thread that is running the query, and because they are asynchronous, the scan does not block while waiting for them to complete. The scan blocks only when it actually tries to scan a page that it thinks has been brought into cache and the read has not yet completed.

- Scan Count

  The Scan Count value indicates the number of times that the corresponding table was accessed. Outer tables of a nested loop join have a Scan Count of 1. For inner tables, the Scan Count might be the number of times ***through the loop*** that the table was accessed. The number of Logical Reads is the summation of the number of pages accessed on each scan. However, even for nested loop joins, the Scan Count for the inner table might show up as 1. SQL Server might copy the needed rows from the inner table into a worktable in cache and use this worktable to access the actual data rows. When this step is used in the plan, there is often no indication of it in the STATISTICS IO output. You must use the output from STATISTICS TIME, as well as information about the actual processing plan used to determine the actual work involved in running a query. Hash joins and merge joins usually show the Scan Count as 1 for both tables involved in the join, but these types of joins can involve substantially more memory. You can inspect the memusage value in sys.sysprocesses while the query is being executed, but unlike the physical_io value, this is not a cumulative counter and is valid only for the currently running query. Once a query finishes, there is no way to determine how much memory it used.

- LOB Reads

  LOB Reads are the logical, physical, and read ahead reads on text, ntext, image or varhcar(max), nvarchar(max), varbinary(max) columns.

## STATISTICS TIME

The output of SET STATISTICS TIME ON shows the elapsed and CPU time required to process the query. The time is separated into two parts:

- The time required to parse and compile the query.
- The time required to run the query.

In many cases, the output includes two sets of data for the parse and compile time. This happens when the plan is being added to syscacheobjects for possible reuse. The first line is the actual parse and compile, before the plan was placed in cache, and the second line appears when SQL Server is retrieving the plan from cache. Subsequent executions will still show the same two lines, but if the plan is actually being reused, the parse and compile times for both lines will be 0. If the query runs with parallelism, the CPU value

shows the cumulative amount of time spent by all threads. So, it is possible for the CPU time to be greater than the elapsed time.

### SHOWPLAN_XML and STATISTICS XML

The big difference between SHOWPLAN_XML and STATISTICS XML is that SHOWPLAN_XML does not actually run the query. Therefore, SHOWPLAN_XML will not have the <**RunTimeInformation**> element. The XML query plans will not show how many reads occurred (logical, physical, etc.), or the total execution time. Therefore, in order to get execution times and reads, you must still enable SET STATISTICS TIME ON and SET STATISTICS IO ON. You can save the XML query plan as a **.sqlplan**, which you can then use to view the query plan, in a graphical format, by using SQL Server Management Studio.

> **Note**: SQL Server 2005 will not allow you to capture any type of showplan for an encrypted object (for example, a stored procedure or a function). If SHOWPLAN was enabled, the server will not produce any trace events, and you will not get any showplan result to the client.

You can expand and collapse the XML output tags. By default, the entire query plan is expanded. If the batch contains several statements, it generates one XML node per statement and concatenates them together.

The following table describes the contents of various XML output tags:

| XML Output Tag | Contains… |
|---|---|
| <ShowplanXML> | The root element. It contains the locations of the Showplan XML schema and the SQL Server build information. |
| <Statements> | The sub-elements of SHOWPLAN_XML. |
| <StatementText> | The attribute of the Statements element. It describes the Transact-SQL (T-SQL) query that is being executed. |
| <StatementSetOptions> | The attribute of the Statements element. Contains SET options that were set when the query was executed. |
| <QueryPlan> | The beginning of the query plan. This node contains plan information. |
| <CachedPlanSize> | The attribute of the QueryPlan node. It shows how much memory space (in KB) the plan takes up in cache. |
| <DegreeofParallelism> | The attribute of the QueryPlan node. You can use the DegreeofParallelism node to determine whether the query is parallel. This attribute is only available when using STATISTICS XML. It does not appear in the SHOWPLAN_XML output. |
| <MissingIndexes> | The element of the QueryPlan node. If present, it indicates that the query optimizer has suggested implementing an index to improve the query performance. |
| <RelOp NodeId = ……> | The element of the QueryPlan node. It shows each execution step. Each <RelOp> element is equivalent to a single row from a SQL Server 2000 showplan. Many of the common columns (such as **EstimateRows** and **EstimatedTotalSubtreeCost**) that you would be interested in are attributes of the <**RelOp**> element. An execution step can be a join operation, an index seek or scan, etc. The <**RelOp**> node |

| XML Output Tag | Contains… |
|---|---|
| | also contains operator-specific information as subelements. |
| <RunTimeInformation> | The sub-element of the RelOp node. It shows the actual number of rows returned, number of executions, etc. This node tag is only available with SET STATISTICS XML and not with SHOWPLAN_XML. |
| <ParameterList> | That attributes that show the value for a parameter, with which the procedure was compiled, and the value for the parameter when the procedure is run. |
| <CachedPlanSize> | The attribute that shows how much memory (in KB) the plan takes up in cache. |

The various attributes of the **RelOp** node tag are:

- **EstimateRows**: This appears in the RelOp node and specifies the number of rows that the query optimizer estimates to be returned by the RelOp operator in the query plan. If the operator will be run more than once during the query, then EstimateRows defines the estimated number of rows to be produced by each execution of the operator. EstimateRows is the column that shows the cardinality estimates for the query.
- **EstimatedTotalSubtreeCost**: This appears in the RelOp node and specifies the estimated cost of the RelOp operator and all operators below it in the tree. Each operator calculates its estimated cost based on the number of rows to process, row size, the number of I/Os that are anticipated to be required to do the work, etc. For the purposes of estimating the cost, the query optimizer assumes that no data is in cache at the beginning of the query; if a page is touched, it will require a physical disk I/O, unless it was read earlier during the processing of the query.
- 

  The cost estimate bears some resemblance to clock time. On the machine on which the costs were calibrated, a cost of 1 corresponds to a 1-second run time for the query. That was several years ago when a Pentium chip and a single SCSI drive was used. The actual execution time (in seconds) for many servers today is quite often much faster than the estimated cost, because many pages may already be in cache and processor speeds and I/O speeds are faster, etc. However, it is still relevant to determine whether one query would be faster than another by looking at the **EstimatedCost** column for the root node of the tree. In fact, the query optimizer picks the plan with the lowest estimated cost; it does not necessarily pick the one that is estimated to have the fewest I/Os. While the number of I/Os that a query takes certainly is factored into the cost, the relative amount of CPU required to process a row is also considered. The end goal of the query optimizer is to return the results to the user in the shortest amount of time.
- **EstimatedRewinds** and **EstimatedRebinds**: This appears in the RelOp node. XML showplan display a count of estimated rewinds and rebinds, instead of the familiar Executes column from older showplans. Previous versions of SQL Server have

calculated and used rewinds and rebinds internally. The Executes column is the sum of rewinds plus rebinds.

If the RelOp is a join, then the next RelOps in the output with higher NodeIds are the tables being joined.

In the example shown below, NodeId = 0 indicates a nested loops join. So, you can find the tables being joined in the <OutptutList> section under RelOp NodId =2 and RelOp NodeId= 3.

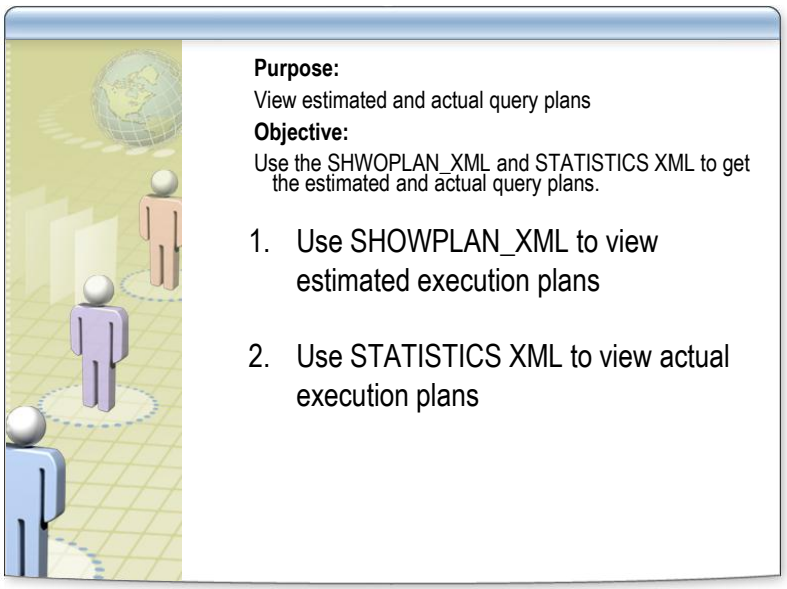
```
<RelOp NodeId="0" PhysicalOp="Nested Loops" LogicalOp="Inner Join"
EstimateRows="290" EstimateIO="0" EstimateCPU="0.0012122" AvgRowSize="4341"
EstimatedTotalSubtreeCost="0.10533" Parallel="0" EstimateRebinds="0"
EstimateRewinds="0">

<RelOp NodeId="2" PhysicalOp="Clustered Index Scan" LogicalOp="Clustered Index
Scan" EstimateRows="290" EstimateIO="0.00756944" EstimateCPU="0.000476"
AvgRowSize="69" EstimatedTotalSubtreeCost="0.00804544" Parallel="0"
EstimateRebinds="0" EstimateRewinds="0">

<RelOp NodeId="3" PhysicalOp="Clustered Index Seek" LogicalOp="Clustered Index
Seek" EstimateRows="1" EstimateIO="0.003125" EstimateCPU="0.0001581"
AvgRowSize="4285" EstimatedTotalSubtreeCost="0.0960728" Parallel="0"
EstimateRebinds="289" EstimateRewinds="0">
```

## Graphical Query Plans

You can view the execution plan graphically. View the estimated execution plan in the Query window by choosing "Display Estimated Execution Plan" from the Query menu. If you choose the estimated execution plan, the query is not executed. The estimated plan is useful to get a quick idea of what the SQL optimizer estimates will be the best execution plan based on the current state of statistics. This can be helpful when developing a new query in a controlled environment where the state of the statistics is known. However, you should not use the estimated query plan as a definitive view of a query execution when troubleshooting performance problems on a production system.

To view the actual execution plan in a Query window, click "Include Actual Execution Plan" on the Query menu. When you execute the query, the actual execution plan will be displayed in the Execution Plan tab in the Query results window.

## Demonstration 3: Viewing the Execution Plan




1. Execute the following batch:

```
use adventureworks
go
set statistics xml on
go
select a.productid, orderqty, listprice, class, color
from Sales.SalesOrderDetail a
join  Production.Product b  on a.ProductID =b.ProductID
where OrderQty >4
```

2. Double click on the XML tag in the results window. If you are using SQL 2005 Management Studio this should bring up the XML query plan in another query window. If you are using SQL 2008 Management Studio this will bring up the graphical query plan in another window.
3. Rename the file save in step 3 to use a .SQLPLAN extension. For example if the file name was query 1.xml it should now be query 1.sqlplan. Once the extension has been renamed, double click on the file. This should bring up the graphical execution plan in SQL Server Management Studio. This step is needed if you are using SQL Server 2005 Management Studio.
4. Rename the file save in step 3 to use a .SQLPLAN extension. For example if the file name was query 1.xml it should now be query 1.sqlplan. Once the extension has been renamed, double click on the file. This should bring up the graphical execution plan in SQL Server Management Studio.

You can use SET STATISTICS XML or SET SHOWPLAN_XML to view actual or estimated query execution plans. The following examples help you compare and contrast these two methods.

You can use SHOWPLAN_XML as follows:

```
Use Adventureworkspto
Go
set showplan_xml on
go
select firstname, emailaddress
from person.contact
where lastname >'mary'
go
set showplan_xml off
```

Or to get the actual execution plan after retrieving the results, use STATISTICS XML as follows:

```
Use Adventureworkspto
Go
set statistics xml on
go
select firstname, emailaddress
from person.contact
where lastname >'mary'
go
set statistics xml off
```

You can also use query plans to view information about joins as follows:

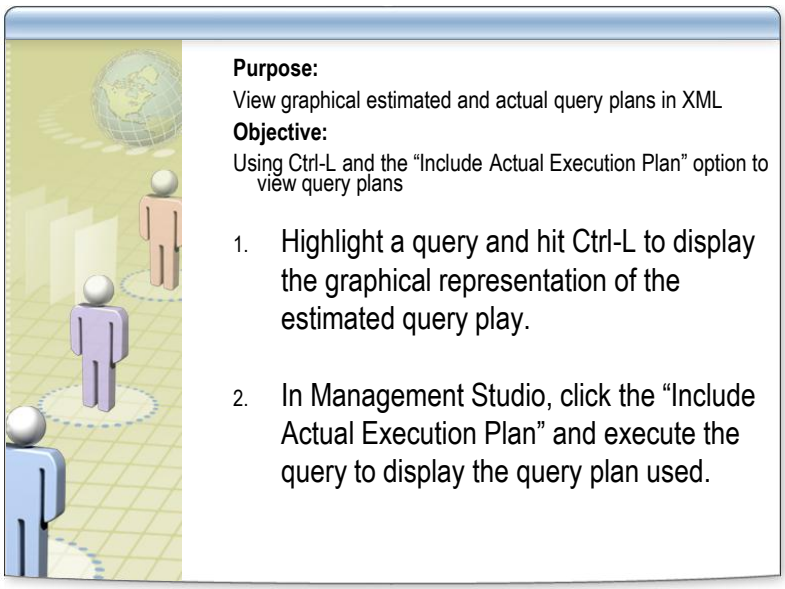
```
Use adventureworkspto
Go
set showplan_xml on
go

SELECT
    e.[EmployeeID]
    ,c.[Title]
    ,c.[FirstName]
    ,c.[MiddleName]
    ,c.[LastName]
    ,c.[Suffix]
    ,e.[Title] AS [JobTitle]
    ,c.[Phone]
    ,c.[EmailAddress]
    ,c.[EmailPromotion]
    ,c.[AdditionalContactInfo]
FROM [HumanResources].[Employee] e
    INNER JOIN [Person].[Contact] c
    ON c.[ContactID] = e.[ContactID]
go
set showplan_xml off
```

Or get the actual execution plan after retrieving the results, use STATISTICS XML as follows:

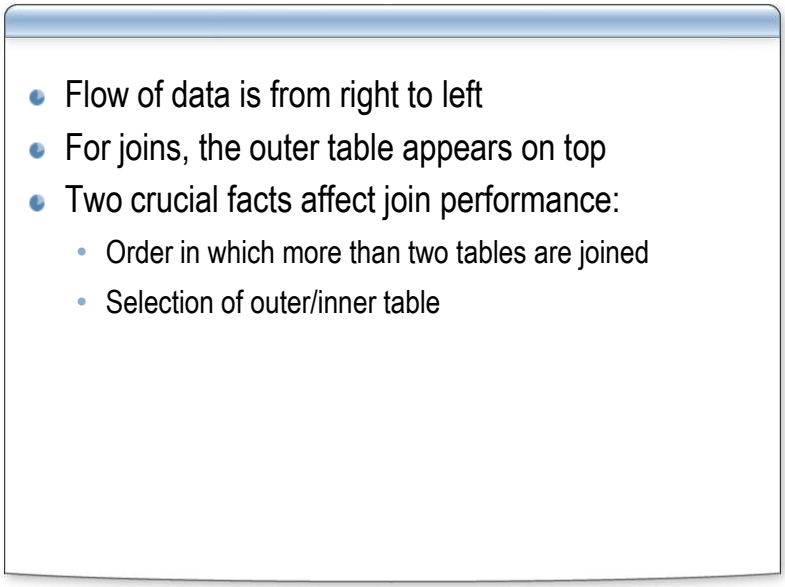
```
Use adventureworkspto
```

```
Go
set statistics xml on
go

SELECT
    e.[EmployeeID]
    ,c.[Title]
    ,c.[FirstName]
    ,c.[MiddleName]
    ,c.[LastName]
    ,c.[Suffix]
    ,e.[Title] AS [JobTitle]
    ,c.[Phone]
    ,c.[EmailAddress]
    ,c.[EmailPromotion]
    ,c.[AdditionalContactInfo]
FROM [HumanResources].[Employee] e
    INNER JOIN [Person].[Contact] c
    ON c.[ContactID] = e.[ContactID]
go
set statistics xml off
```

# Demonstration 4: Viewing graphical query plans

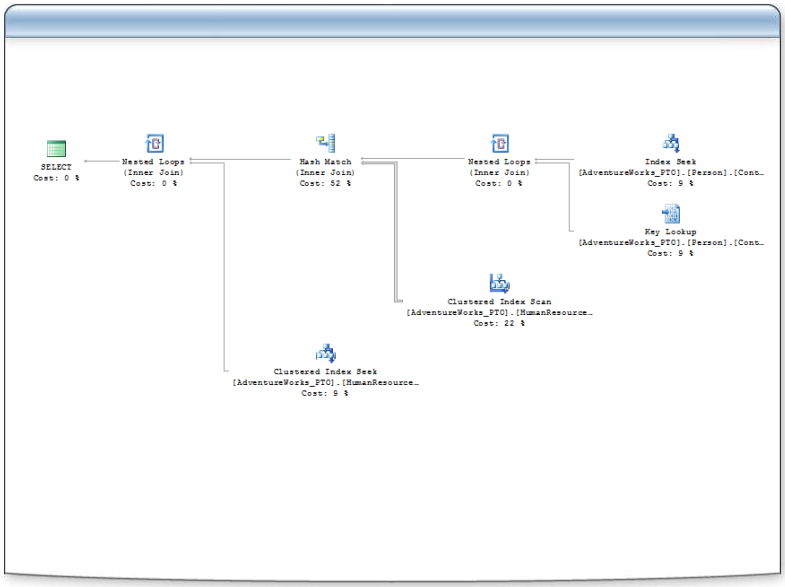


## Joins in Graphical Showplan




The query plan should be read from right to left. If the query joins the tables together, the table listed on top is the outer table, which means that that table would be referenced first. If the query is joining multiple tables together, the query optimizer will join two tables and produce a resultset that is then joined to another table in the query.

# Query Plan Example




24

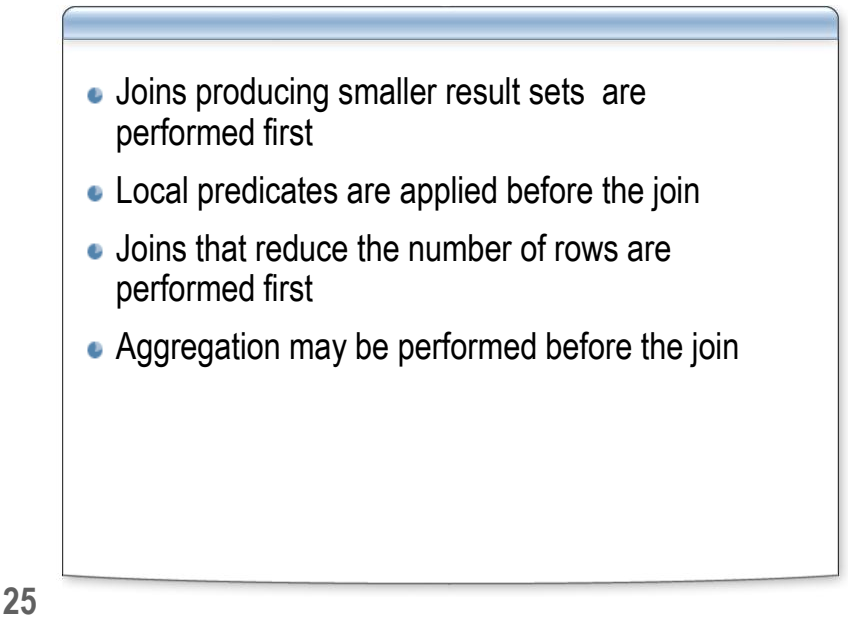
The query plan shown on the slide above was generated from the following Select statement:

```
SELECT EMAILADDRESS, FIRSTNAME, LASTNAME,
B.TITLE,c.modifieddate
FROM PERSON.CONTACT A
JOIN HUMANRESOURCES.EMPLOYEE B
ON A.CONTACTID = B.CONTACTID
join humanresources.employeeaddress c
on b.employeeid=c.employeeid
WHERE EMAILADDRESS = 'RENEE12@ADVENTURE-WORKS.COM'
```

You should read the query plan from right to left and top to bottom. So the **Person.Contact** table nonclustered index was joined to the clustered index on the Person.Contact table. This operation is a lookup, which was known as a *bookmark lookup* in the previous releases. The Person.Contact table was joined to the **HumanResources.Employee** table to produce a resultset by using the hash join algorithm. The resultset was then joined to the **HumanResources.EmployeeAddress** table using the nested loops join algorithm.

To optimize a query, you should review the percentage of time that was spent in each part of the query. In the example shown above, 52 percent of the query execution time was spent performing the join between the Person.Contact table and the HumanResources.Employee table. Therefore, you should optimize the query to see if you can leverage a different join algorithm.

## Multi-Table Joins

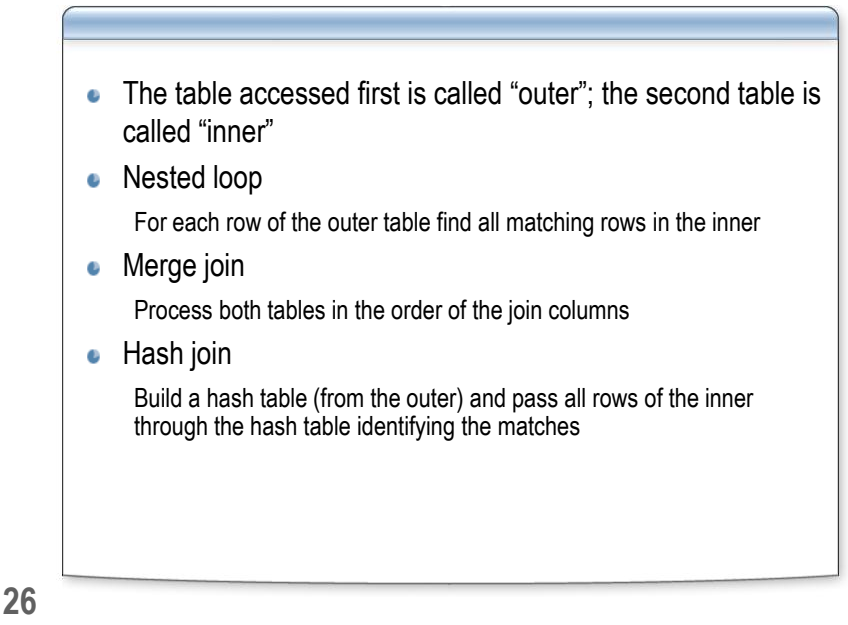



When performing multiple-table joins, the resultset is narrowed down as it progresses from one join to another. Smaller tables are joined first. If the query is joining three tables, the query optimizer might choose to first join the two tables that will reduce the size of the intermediate resultset the most, and then perform subsequent joins.

Local predicates are applied before the join process, so that after the filter is applied, there will be fewer rows to process in the join.

Any aggregation may be performed before the join process. The advantage is that after performing the aggregate, there will be fewer rows that need to be joined to the other table.

# Join Types



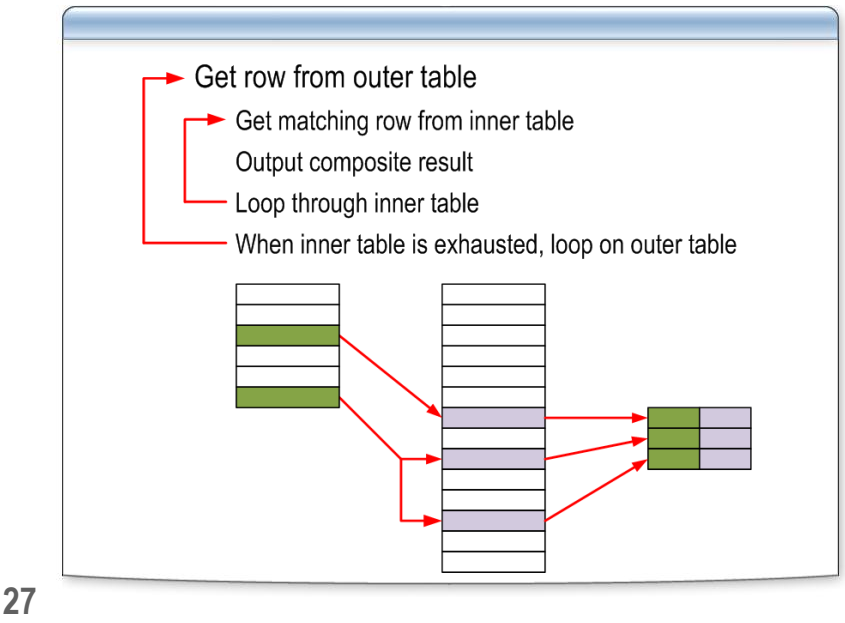

SQL Server has three different join algorithms: nested loop join, merge join, and hash join.

Join algorithms may require the tables to be sorted before they are joined. Therefore, the query optimizer may internally use a work table to sort the tables used in a join.

The resultset can be returned in the sort order if a nested loop or merge join is used. If a hash join was used to produce the resultset, it may not be ordered.

Which join type is more efficient depends on the size of the tables being joined, the distribution of data in that table, and whether or not the data can be retrieved in order. The query optimizer evaluates the relative expense of these algorithms in the light of these data characteristics to determine which join type should be used.

# Nested Loop Join




A nested loop join:

- Finds all matches in the inner table for each row of the outer table.
- Is best used when there is a supporting index on the inner table.
- Has a low memory requirement.
- Requires that the smaller resultset appear as the outer table.

A nested loop join operator has two inputs—an outer table (also known as outer loop) and an inner table (also known as inner loop). The top input to the nested loop join operator is the outer table, and the bottom input to the nested loop join operator is the inner table.

SQL Server may sort the outer table to improve the locality of seeks. A nested loop join takes a row from the outer table and uses the join key values from that row to seek or scan the inner table for matching rows. If any matching rows are found in the inner table, the row is returned.

## Batch sort

**Note**: SQL Server 2005 has a *batch sort optimization* that it may use preceding an operation where sorted data is required, but it is not necessary to retrieve all of the results. For example, when TOP, SET ROWCOUNT or semi-joins are used with a loop join, a batch sort may be used.

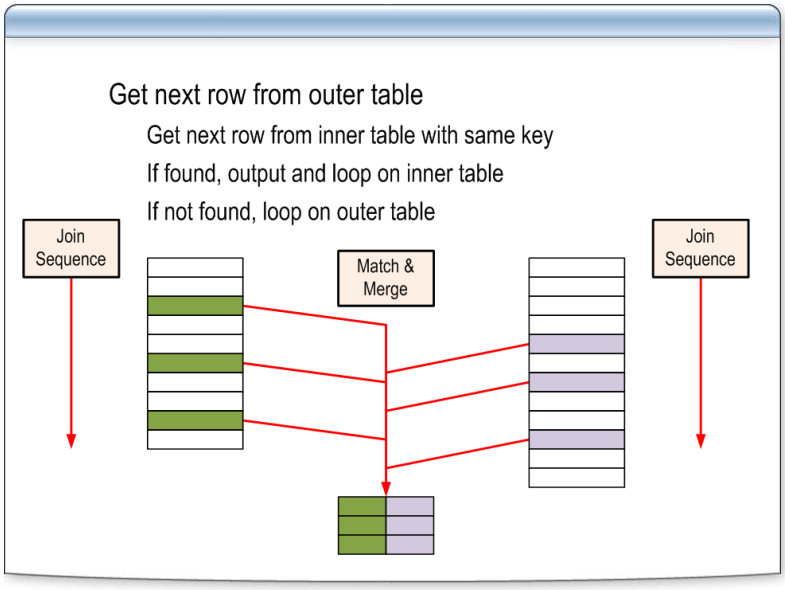
Sometimes, it is beneficial if the columns from the outer table of a nested loop join are sorted before doing the lookups on the inner table. (The lookups traverse forward, through the inner table, and, if there are multiple lookups, SQL Server does not need to perform a physical I/O to read the page back in.) While it can be beneficial to sort the input rows, it also requires that all rows be read and sorted before any of the lookups in the outer table can start (a regular Sort operator reads all of its input, sorts it, and then starts producing the output). If SQL Server needs only a portion of the output (for example, when you are using TOP, SET ROWCOUNT, or semi-joins), then the execution delay from a regular sort is clearly undesirable. The batch sort operator reads a subset of its input, sorts that group of rows, and then returns those rows. If the calling function needs more rows, the batch sort operator reads another group of rows from its input, sorts them, returns the output, etc. In doing so, the batch sort operator allows the server to output data more quickly than it might be able to if all the rows were read and sorted at one time.

You can determine when a batch sort was used if you see the OPTIMIZED keyword in a legacy showplan, or the Optimized="1" attribute in XML showplan for the Nested Loops operator, as shown in the example below:

```
<NestedLoops Optimized="1">
```

A plan with a batch sort requires a memory grant. Therefore, you must be aware of the fact that a plan that only contains loop joins could still, potentially, wait on a RESOURCE_SEMAPHORE waittype if there is insufficient memory for the grant, at that time.

# Merge Join

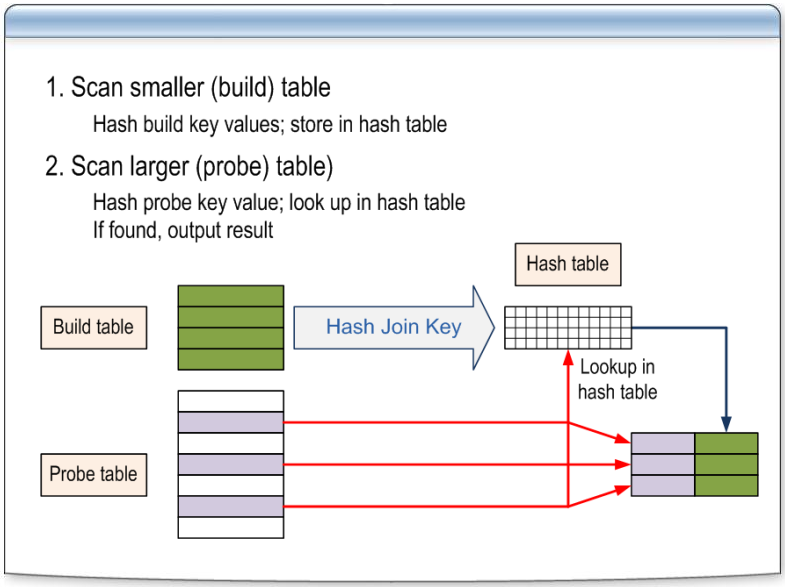



A merge join:

- Requires both inputs to be sorted in the order of the join key when entering the join.
- Uses an index to provide the order if one is available. The optimizer may introduce a sort to perform a merge join if no index is available on the join keys.
- Enables both small and large tables to appear on either side of the join.
- Requires low memory, unless many duplicates occur on the inner side.
- Requires SQL Server to store inner duplicates if there are duplicates on the inner side and there might be duplicates on the outer side. (Inner or outer selection is not as important for a merge join as it is in loop and hash joins.)

A merge join simultaneously passes over two sorted inputs to perform inner joins, outer joins, semi-joins, intersections, and union logical operations. A merge join exploits sorted scans of B-tree indexes and is generally the preferred method if the join fields are indexed, and if the columns represented in the index cover the query.

# Hash Join




A hash join:

- Uses the outer table to build a hash table.
- Completely reads the outer table before accessing the first inner row. (This is unique to hash joins.)
- Uses the smaller table as the outer table.
- Does not preserve the order of either input. (This is unique to hash joins.)
- Requires more memory than other join types. The larger the outer table, the more memory is required to build the hash table.
- Does not use the order provided by indexes.
- Is often the most efficient method of joining data sets with no supporting indexes.

> **Note**: Hash join is based on the ***hashing search algorithm***.

Hash join takes two inputs:

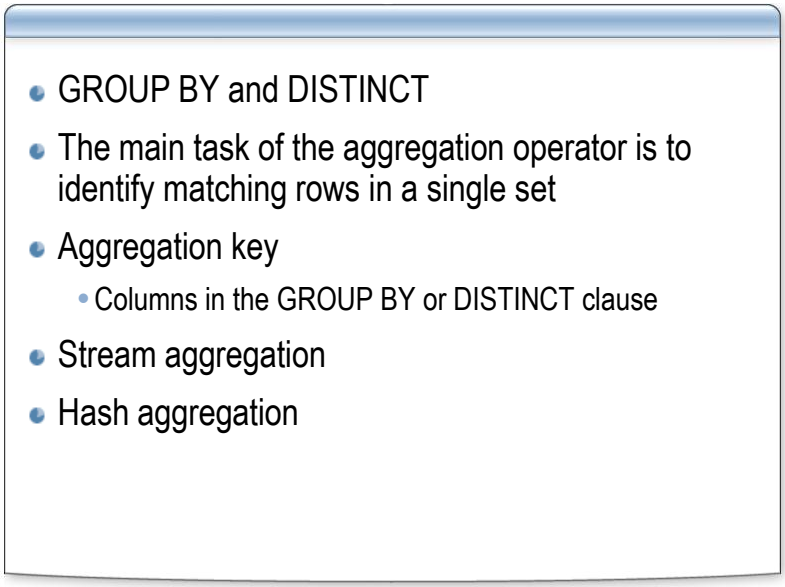
- **Build input**. This input builds hash buckets by using complicated hash functions that group the data. Each row is inserted into a hash bucket, depending on the hash value computed for the hash key. The buckets are stored as linked lists where each entry contains only those columns from the build input that are needed.

- **Probe input**. For each record in the probe input, this input evaluates the hash key and checks the appropriate hash buckets in the linked list (which is built by the build input) for matches and returns matching values.

The build input is completed before the probe is actually read because the hash tables need to be built first. This is different from what happens in a nested loop join.

The query optimizer chooses the smaller of the two tables being joined to be a build table.

# Aggregation

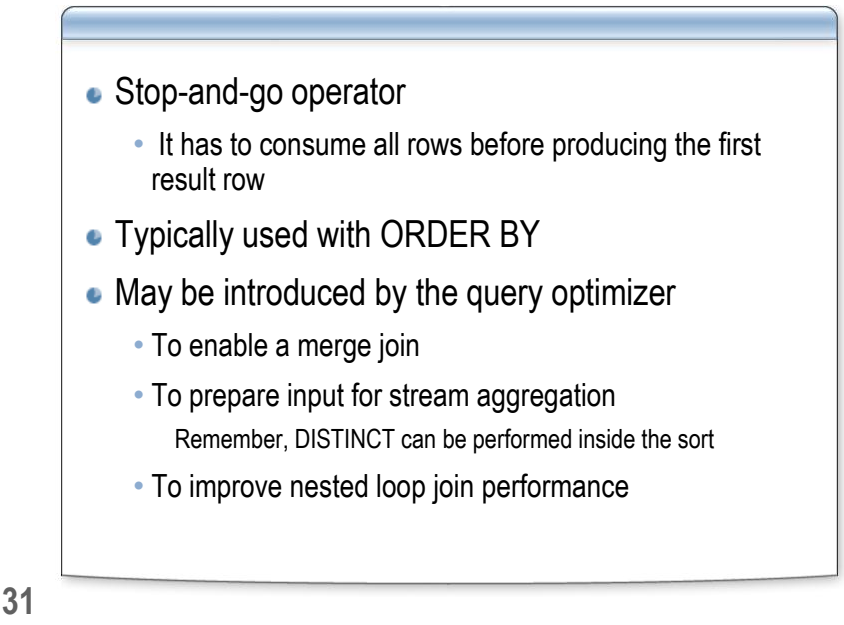



If a query uses a **GROUP BY** or **DISTINCT**, the query optimizer performs an aggregation based on the columns listed in the **GROUP BY** or **DISTINCT** clause. The query optimizer has two algorithms for performing aggregations—stream and hash. The hash aggregation can be more expensive in terms of CPU and memory.

**Stream aggregation** is performed only on sorted data sets. The Stream Aggregate operator groups rows by one or more columns and then calculates one or more aggregate expressions returned by the query. Because the data is sorted, when the value of the **GROUP BY** key changes in the resultset, SQL knows that it is finished with the aggregation of the previous group, and therefore, returns the resulting aggregated row. The optimizer may introduce a sort operator prior to a stream aggregation if the data is not already sorted due to a prior Sort operator or due to an ordered index seek or scan.

**Hash aggregation** is performed on unsorted datasets. Hash aggregations build a hash table and use it to track different values of the **GROUP BY** keys. Hash aggregation also aggregates the entire dataset before returning any aggregated rows.

# Sort

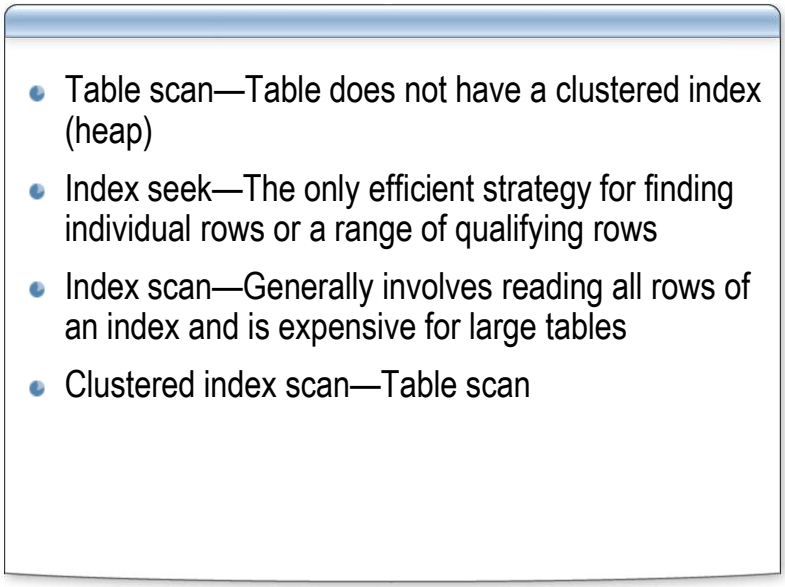



The **Sort** operator sorts all incoming rows. The Argument column contains either a DISTINCT ORDER BY:() predicate if duplicates are removed by this operation, or an ORDER BY:() predicate with a comma-separated list of the columns being sorted. The columns are prefixed with the value **ASC** if the columns are sorted in an ascending order, or the value **DESC** if the columns are sorted in a descending order. A sort operation must consume all rows from the source dataset before any rows can be produced in the result and sorted dataset.

Sorts are typically seen when using an **ORDER BY** clause on a column that does not have an index. However, a **Sort** may also be introduced by the optimizer either to enable a merge join or to prepare for a stream aggregation in cases where the introduced sort, and the merge join or stream aggregation together may be cheaper than the alternatives with unordered datasets.

> **Note**: Sort operations are efficient with small datasets to be sorted, but become increasingly inefficient as the size of the dataset increases. As a result, indexes that support ORDER BY clauses become more important as the data in a table increases in size. In addition, the optimizer is less likely to introduce a sort operation on larger datasets for the purpose of enabling a stream aggregation or a merge join.

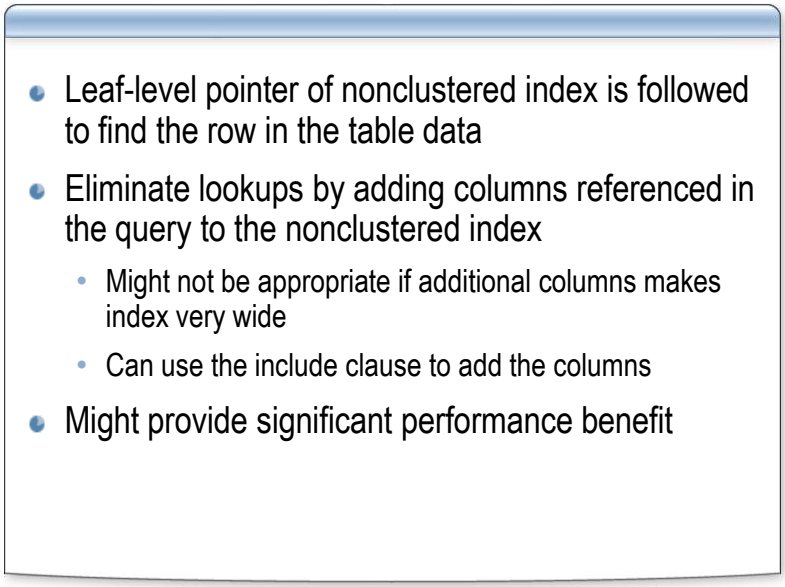
## Scan and Seek Operators




Scans generally consume more IO than seeks.

If the query optimizer uses a clustered index scan or clustered index seek, it implies that the clustered index is used. If the query plan shows an index scan or index seek, it implies that a non-clustered index was used.

## Lookups




A lookup is the same as a bookmark lookup in earlier editions of SQL Server. A lookup happens when SQL Server uses a non-clustered index to access a row and then must lookup the row in the base table to retrieve other columns needed by the query (referenced in either the select list, join clause, or where clause) that are not part of the non-clustered index. Consequently, a lookup incurs additional I/O and CPU cost.

Suppose your query is retrieving COL1, which defines Index1, but your predicate uses COL2, on which an index called Index2 is defined. The following steps will be performed as part of a lookup:

1. SQL Server performs an index seek on Index2 to find the rows that fit the first criteria in the query.
2. If a clustered index exists on this table, then the clustered index keys are contained in the non-clustered index. If COL1 is not a clustered index key, and is not an included column, then SQL Server uses a nested loop operator to join to the clustered index and seeks the row in the clustered index to retrieve COL1. This will appear in the execution plan as a **KEY LOOKUP**.
3. If no clustered index exists on the table, then a Row Identifier (RID) is contained at the leaf level of the non-clustered index. The RID contains the FileID, PageID, and SlotID where the row can be found in the base table. SQL Server uses a nested loop to access this page directly to retrieve the value for COL1. This operation appears in the query play as a **RID LOOKUP**.

In the XML execution plan, the column looked up will appear both in the column reference tag of the output list of the nested loop operation and the column reference tag of the lookup operation node.

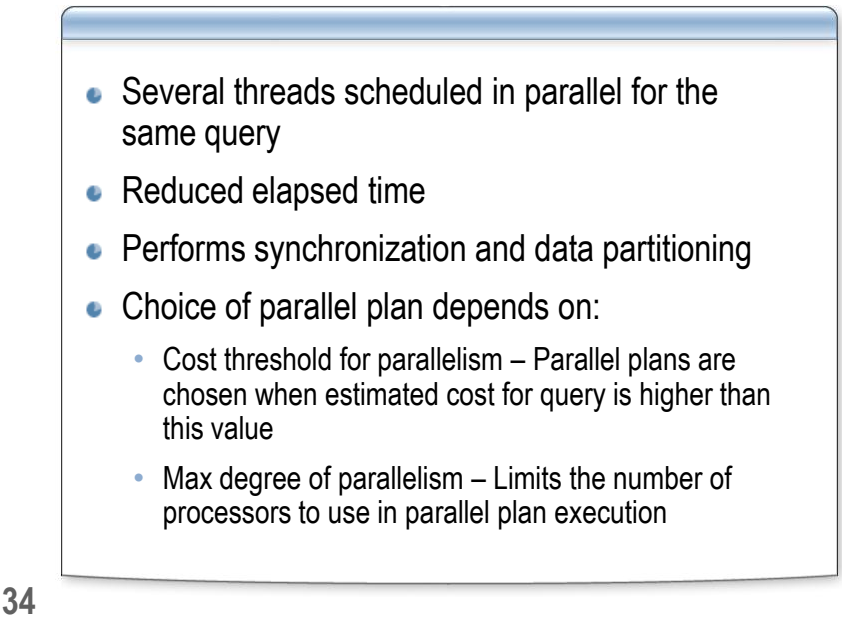
You can remove lookups by using ***covering columns***. The list of covering columns includes the non-clustered index keys, the clustered index keys, and any columns included in the index definition. Covering columns enables you to include the referenced columns in the non-clustered index.

Adding columns to an index may not be appropriate if the additional columns make the index too wide. Wide indexes will require more maintenance on every INSERT, UPDATE, or DELETE, and will require more processing during normal index maintenance.

You can add columns as included columns on an index to eliminate lookups while minimizing the size of the non-leaf levels of the index.

Because lookups themselves are expensive operations, including columns to eliminate lookups can provide significant performance benefits to select queries. When large numbers of rows are returned, it may be cheaper to perform a table or clustered index scan than to perform an index seek with the large numbers of lookup. In such cases, including columns to eliminate the lookups can actually eliminate large scans, and therefore, have a large performance benefit.

# Parallelism




SQL Server 7.0 introduced *intra-query parallelism*, or the ability to break a single query into multiple subtasks and distribute them across multiple processors in a multi-processor machine for execution.

The architecture uses a generic parallelization operation that creates multiple, parallel threads, when needed. Each operation (scan, sort, and join) is unaware of parallelism but can be executed in parallel, simply because it is combined with the parallelization operation, resulting in ***parallel everything***. The exception to parallel everything is *parallel updates*, which Microsoft will add in a future release of SQL Server. All other operations (scans, sorts, joins, GROUP BYs) can be done in parallel.

> **Note**: Although the update itself cannot be performed in parallel, a scan to locate one or multiple rows to be updated can be performed in parallel. Therefore, update queries can have parallel execution plans.

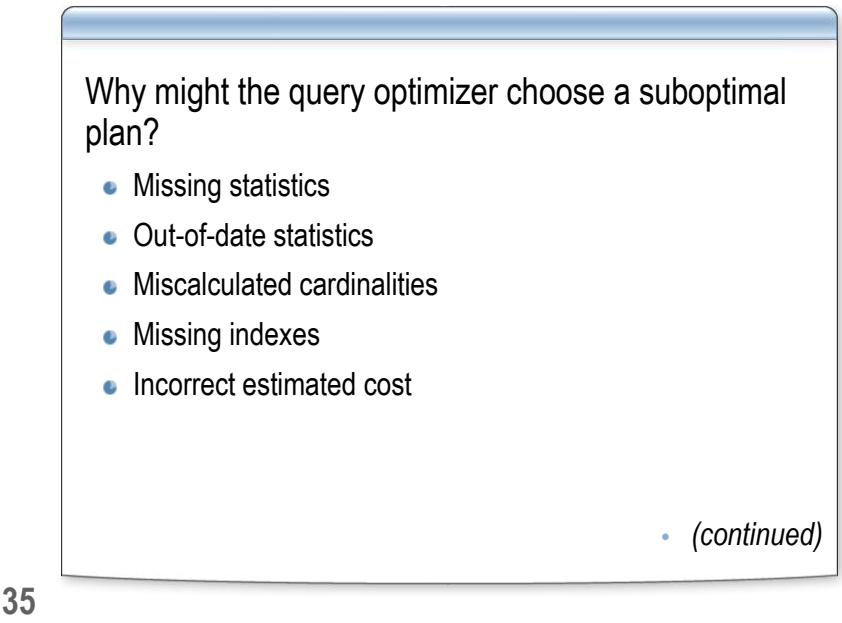
SQL Server detects that it is running on an SMP machine and determines the best degree of parallelism for each instance of a parallel query execution. By examining the current system workload and configuration, SQL Server determines the optimal number of threads and spreads the parallel query execution across those threads. Once a query starts running, it uses the same number of threads until completion. SQL Server decides the optimal number of threads each time a parallel query execution plan is retrieved from the procedure cache. As a result, one execution of a query can use a single thread, and another execution (at a different time) of the same query can use two or more threads.

## Factors affecting parallelism

The decision about whether to employ parallel query processing in SQL Server depends on the answers to the following questions:

| Question | Answer |
|---|---|
| Is SQL Server running on a computer with more than one processor? | Only computers with more than one processor can take advantage of parallel queries. |
| Which edition of SQL Server 2005 or 2008 are you running and whether you are using the 32 bit or 64 bit? | To determine the number of processors supported, refer to the topic *Maximum Number of Processors Supported by the Editions of SQL Server 2005* in SQL Server Books Online. |
| What is the number of concurrent users active on SQL Server? | SQL Server monitors CPU usage and adjusts the degree of parallelism at query startup time. Lower degrees of parallelism are chosen if the CPUs are busy. |
| Is there sufficient memory available for parallel query execution? | Each query requires a certain amount of memory to run. Parallel queries require more memory than nonparallel queries. The amount of memory required for running a parallel query increases with the degree of parallelism. If the memory requirement of the parallel plan for a given degree of parallelism cannot be satisfied, SQL Server automatically decreases the degree of parallelism, or completely abandons the parallel plan for the query in the given workload context, and runs the serial plan. |
| What is the type of query being executed? | Queries consuming many CPU cycles are the best candidates for a parallel query. Examples include joins of large tables, substantial aggregations, and sorts of large resultsets. Simple queries, often found in transaction processing applications, find that the additional coordination required to run in parallel, outweighs the potential performance boost. To distinguish between queries that benefit from parallelism and those that do not, SQL Server compares the estimated cost of running the query with the cost threshold for parallelism. Although we do not recommend it, administrators can change the default cost threshold. |
| Are there sufficient rows in the given stream? | If the query optimizer determines that the number of rows in a stream is too low, it does not introduce parallel operators. Running a serial plan avoids scenarios where the startup, distribution, and coordination costs exceed the gains achieved by parallel execution. |

## Suboptimal Plans




General estimation issues are:

- **Missing statistics**

  The histogram on an index is maintained on the first column of the index only. If a query contains only the first column of this index in the WHERE clause, then this is adequate. But consider a scenario where table tab1 has an index on col1 and col2. In this scenario, if you run the following query:

```
SELECT * FROM tab1 WHERE col1 BETWEEN 1 AND 100 AND col2 = 2000;
```

  Then, you need to understand that there is no histogram that allows SQL Server to estimate the number of rows where col2 = 2000. In order to accurately estimate the number of rows being returned by this combination of col1 and col2, SQL Server needs a histogram on col2.

  If another index exists with col2 as its first column, then SQL Server can use the histogram in the statistics for the other index. If no other index exists with col2 as its first column, then SQL Server needs to create statistics on this subsequent column to be able to have a histogram on which to base its estimate.

  The creation of statistics on col2 is done automatically if AUTO_CREATE_STATISTICS is enabled on the database. However, if this option is not enabled, then the query plan will show the **missing statistics** warning. In this case, SQL must use densities to estimate the number of rows to be returned from the

seek or scan operation, but these densities will not have any information about the ranges of values that exist in the second column.

Enabling AUTO_CREATE_STATISTICS allows SQL Server to build system statistics in such cases, and also allows SQL to better estimate the number of rows returned by scans or seeks of composite indexes. Disabling this option can lead to suboptimal query plans due to missing statistics.

- **Out-of-date statistics**

  Changes in the data as a result of updates, deletes, or inserts can cause the distribution to change from what is represented in the histogram. This data can change densities in certain ranges of data which can change the optimal strategy for access for that range. To understand this better, consider the following example.

  If you have an index on an identity column col1 of a table named tab1, and the greatest value in this 1,000,000 at the time statistics are updated, then the greatest value represented in the histogram is 1,000,000. If you begin inserting rows in tab1, then values will be added above 1,000,000. If inserts are the only modification of data in this table, then you can insert about 200,000 rows before an auto update of statistics occurs. This means that in the range above what is covered by the histogram, you have 200,000 rows.

  If you query for a range of values above 1,000,000, SQL cannot estimate the number of rows that will be returned from that range. In this case, SQL will always estimate one row from that range, and will optimize for one row. In cases where a loop join is used after this index seek, or in cases where a lookup is required, optimizing for one row may produce a very different plan from one optimized for 200,000 rows. In this case, the statistics need to be updated, so that SQL can estimate the number of rows more accurately and choose a more optimal plan.

- **Miscalculated cardinalities**

  Cardinality refers to the total number of rows processed at each level of a query plan. SQL must be able to estimate the cardinality of each level with a high degree of accuracy in order to accurately estimate the cost associated with subsequent operations, and choose an optimal strategy for processing the query. Miscalculating the cardinality—especially when it is underestimated—can lead to explosions in the cost of the query. Miscalculating cardinality can also cause SQL Server to incorrectly estimate whether or not a query can benefit from parallelism. Some factors that can lead to miscalculated cardinality include:

  - Missing statistics.
  - Out-of-date statistics.
  - Use of the **LIKE** operator with a leading **%** in the WHERE or JOIN clauses. When LIKE is used in this manner, SQL Server may be unable to use statistics to

estimate the number of rows that will be returned by a query. In this situation, the SQL Server estimate will not accurately reflect the number of rows returned by the seek or scan operator, and the resultant incorrect estimate of cardinality can cause poor decisions of join or access strategies elsewhere in the query plan.

- Use of **CASE** logic in the WHERE or JOIN clauses. Consider the following query:

```
SELECT * FROM forTestNoAutoCreateStats
WHERE col1 = CASE WHEN @val1 IS NULL then @val2
        ELSE @val1 END;
```

SQL Server is often unable to pre-determine the values for which the query should be optimized. Often, this leads to an incorrect estimate of cardinality, and a resultant poor query plan. SQL will likely optimize for one row in this case. This can cause an explosion of the query cost if (for example) several thousand rows are returned, and subsequently processed through a lookup and several nested loop operations.

- Any other operation in the WHERE or JOIN clauses that does not allow SQL to make use of statistics to estimate the number of rows returned.

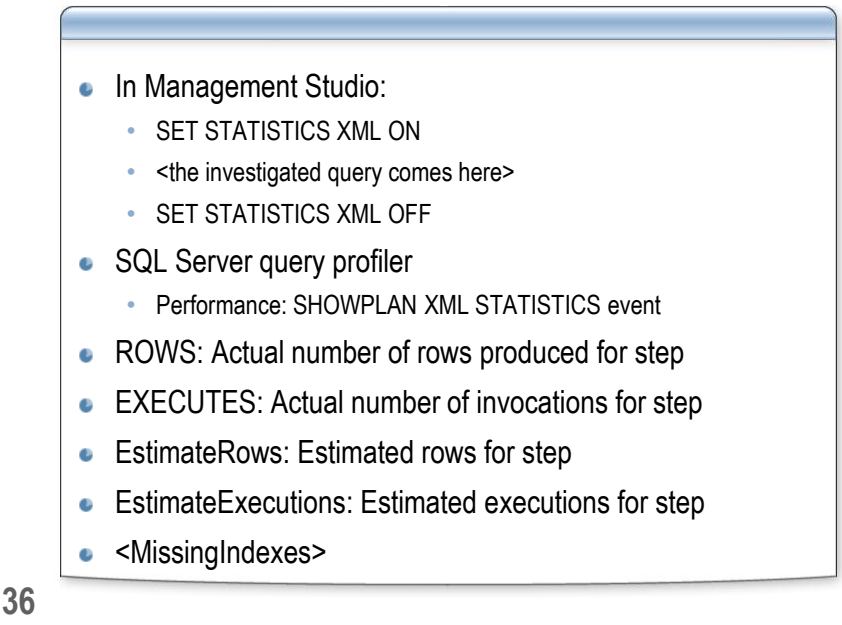
- **Missing indexes**

  If a query retrieves one row from a table of one million rows, it can most efficiently access this one row by navigating the levels of an index to find that row. This will typically allow SQL Server to return this one row with three or four page reads. If no index exists, then SQL must scan the table or index to find the one row that it needs. This will require many more page reads, and an evaluation of all one million rows. For reasons and queries like this, an optimal indexing strategy is the key for producing optimal query plans.

- **Incorrectly estimated cost of the operator**

  One of the more common reasons for an incorrect estimation of the operator cost is an incorrectly estimated cardinality in a prior operation in a query plan. For example, if the LIKE operator is used in a query, SQL Server may not be able to accurately estimate the number of rows returned by the scan on the table where the LIKE operator is used. Suppose that because of its inability to accurately estimate the number of rows, SQL estimates that the scan will return one row. If the query plan involves several joins in addition to the LIKE condition, then SQL will need to determine the join strategy to be used. A nested loop may be the most efficient for one row, so SQL may choose it. But if the number of rows meeting the condition is actually very large, then SQL will incorrectly estimate the cost of a nested loop because of the preceding incorrect estimation of cardinality. This incorrect estimate will be multiplied on every subsequent operation in the query plan.

# Suboptimal Plans (continued)




The query processor derives estimates by using available statistics.

### Automatic statistics creation and refresh

When you create an index, SQL Server automatically stores statistical information regarding the distribution of values in the first indexed column. SQL Server also supports column statistics on subsequent indexed and non-indexed columns. The query optimizer uses these statistics to estimate the size of intermediate query results, as well as the cost of using the index for a query.

If the query optimizer determines that the important statistics for optimizing a query are missing, it automatically creates the required statistics. Such automatically created statistics are saved in the database and can be used to optimize other queries. Moreover, SQL Server automatically updates statistical information, as the data in a table changes and these statistics become out-of-date.

Statistics are created and refreshed very efficiently by sampling. The sampling is random across data pages and is taken either from a table or from a non-clustered index, for the smallest index that contains the columns needed by the statistics. The volume of data in the table and the amount of changing data determine the frequency at which the statistical information is updated. For example, the statistics for a table containing 10,000 rows might need updating when 1,000 rows have changed, because 1,000 is a significant percentage of the table. However, for a table containing 10 million rows, 1,000 changes are less significant and might not trigger an update.

The **EstimateRows** option is per single execution, so you have to multiply it by EstimateExecutions before comparing the result to the ROWS value as follows:

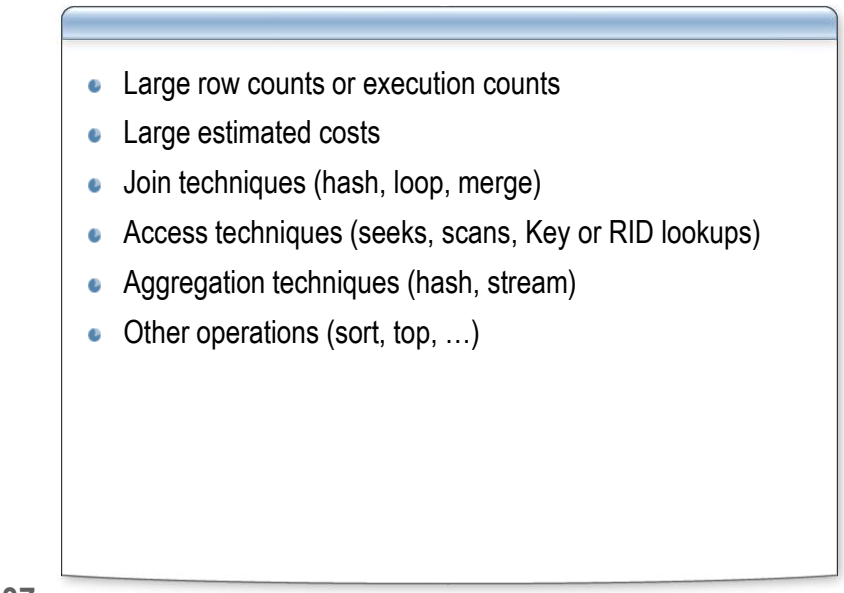
```
Rows = EstimateRows * EstimateExcecutions
```

## Indicators of out-of-date Statistics

When you analyze a query plan, you should compare the actual number of rows and executes (Columns EstimateRows and EstimateExecutions in the Showplan), with those that were estimated.

Substantial differences in the estimated row count might indicate that the query optimizer had out-of-date or skewed statistics. For example, if the estimated row count is 2 rows, and the actual row count is 50,000, the query optimizer may have used either out-of-date or skewed statistics. Try using the UPDATE STATISTICS WITH FULLSCAN command to correct this problem.

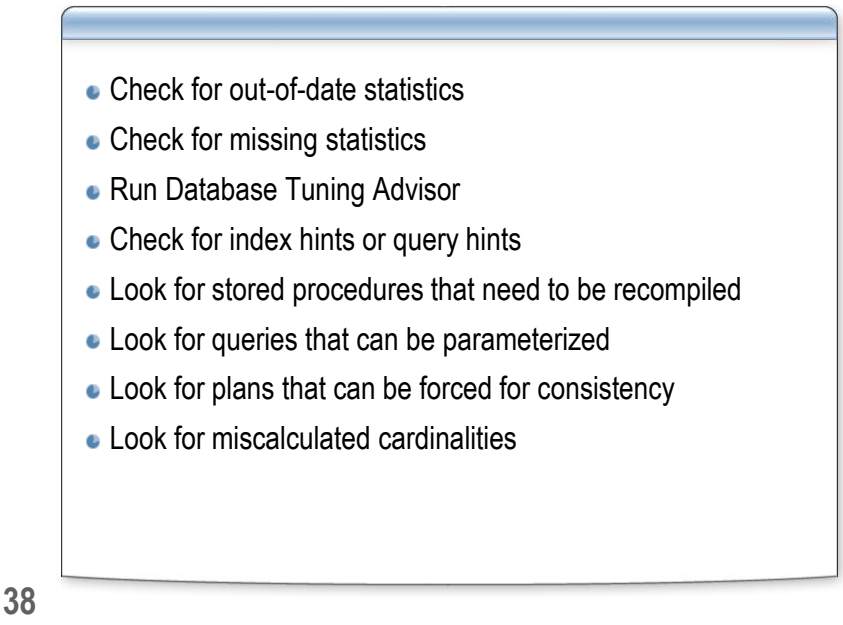
# What Else Should You Look For?




Some queries are expensive either because of the size of the query, or due to missing indexes. Look at expensive nodes in the query plan to find the reasons they are expensive, and analyze what can be done to change the plan. For example, a hash join may be changed to a less expensive operation by the addition of a covering index.

Some operations are inherently expensive. Extract Transform and Load (ETL) queries are often performed to extract large datasets. Such queries expect a large number of rows. Analyze the query plans with an understanding of the extraction requirements.

## Steps to Optimize Queries




Use the following steps when analyzing a showplan:

- Are the statistics for all indexes up-to-date? You should be especially careful of the new NORECOMPUTE clause, which will prevent AUTO_UPDATE_STATISTICS from maintaining statistics.

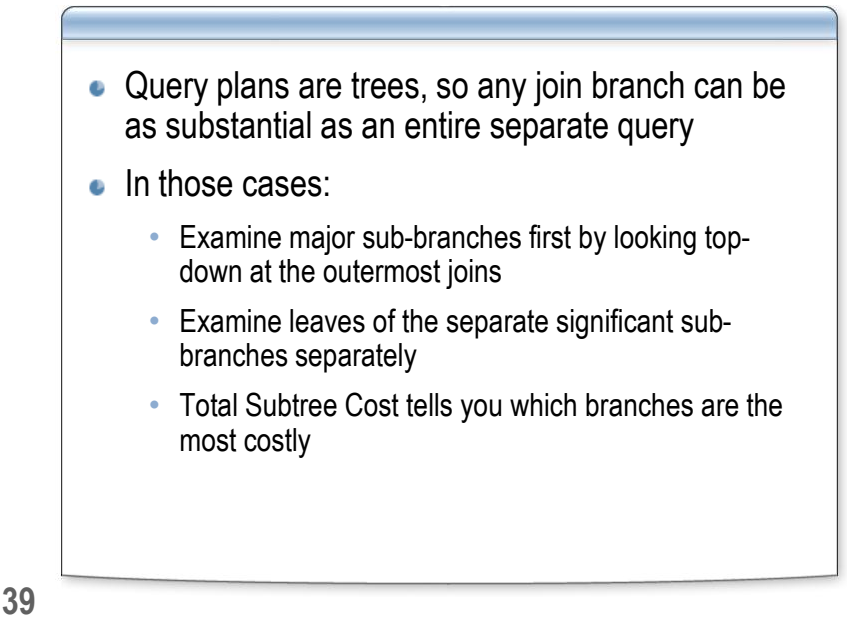
  Following is a sample query using the STATS_DATE function that shows the last time each index on a user table was updated:

```
SELECT o.object_id, s.stats_id,
SCHEMA_NAME(o.schema_id) + '.' + o.name AS [object],
s.name AS [index or stats name],
STATS_DATE(s.object_id, s.stats_id) as [stats last updated]
FROM sys.objects o JOIN sys.stats s ON o.object_id = s.object_id
```

- Look for the ***Missing Stats*** warnings in Profiler or SHOWPLAN_XML output. Consider either creating indexes or allowing **AUTO_CREATE_STATISTICS** in the database.
- Run the Database Engine Tuning Advisor (DTA). Given the changes to the way indexes are stored and manipulated, the indexes that you had on the previous version may no longer be optimal. You should not discount the power of using the DTA.
- Are any index hints or query hints being used? If yes, remember:
  - Any plan containing hints will not be auto-parameterized and cached.
  - Hints prevent the query optimizer from investigating a better alternative.

    - Using a join hint prevents the query optimizer from analyzing optimal join ordering.
- In the case of a stored procedure, are the parameters being passed in such a way that a different plan should be chosen from one iteration to the next? If so, you should consider using WITH RECOMPILE when the procedure is created, or EXEC … WITH RECOMPILE when the procedure with the unusual values is called. A new option since SQL Server 2005 is to modify the query in the stored procedure and use the OPTIMIZE FOR query hint.
- Are there many small, ad hoc queries being used? The best way to handle this is by parameterizing your queries, using parameter markers in OLE DB or ODBC, or using sp_executesql. Use sp_executesql instead of EXECUTE whenever possible, because it avoids recompiles. The server also attempts to minimize the impact by caching plans (this requires fully qualified object names, including the database name) and auto parameterization.
- Can you force a plan that is significantly faster? If the elapsed time to process the query is already very short (less than 250 ms), forcing a plan may just eliminate part of the compile time.
- One of the most common problem areas is incorrect cardinality estimation. Because the cost estimates are based on the cardinality estimates, this can lead to a poor plan decision. To see if this is happening, you should compare the actual execution plan (SET STATISTICS PROFILE ON or SET STATISTICS XML ON). Specifically, compare the estimated rows against the actual rows produced by each operator. You might also observe that the join order changed for certain tables (hash join only), but that generally should not affect performance.

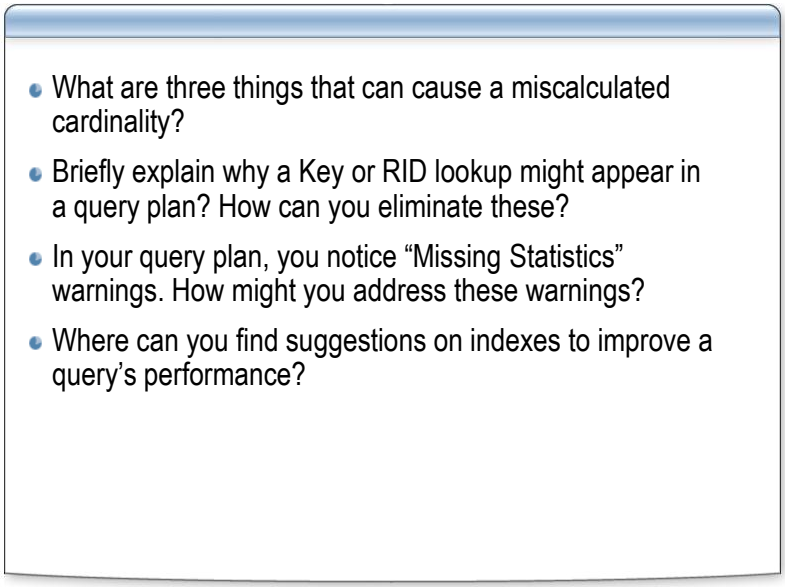
## Complex Queries




Reading complex query plans is not as difficult as it may seem. You must remember that the query optimizer only joins two tables at a time. After it joins these two tables, it joins the resultset to another table.

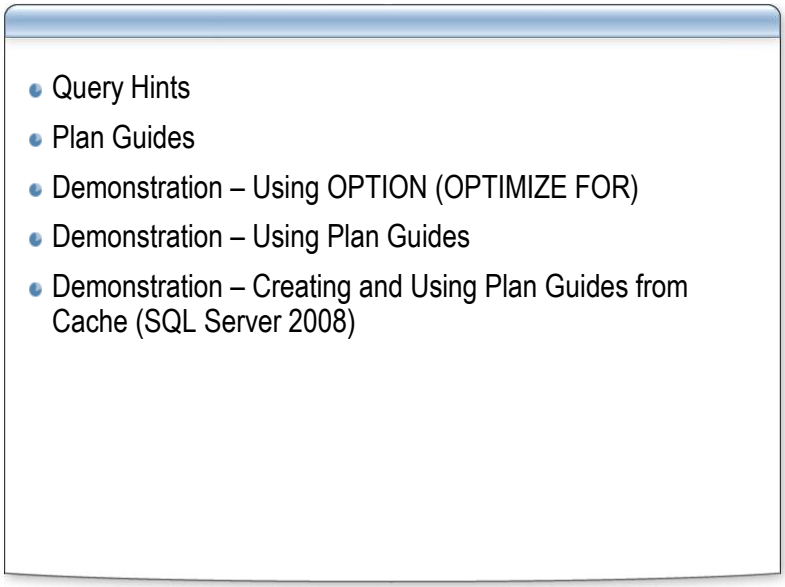
Additionally, when trying to optimize a complex query, you should identify the most expensive part of the query by using either the estimates from the graphical show plan output or the total subtree cost from the text-based query plan, or by reviewing the table that has the most IO associated with it. After identifying the most expensive part of the query, you should try to optimize only that portion of the query and see if the query performance improves.

# Section 3 Review




- What are three things that can cause a miscalculated cardinality?
- Briefly explain why a Key or RID lookup might appear in a query plan. How can you eliminate these?
- In your query plan, you notice ***Missing Statistics*** warnings. How do you address these warnings?
- Where can you find suggestions on indexes to improve the performance of a query?

# Section 4: Creating Plan Guides




## Introduction

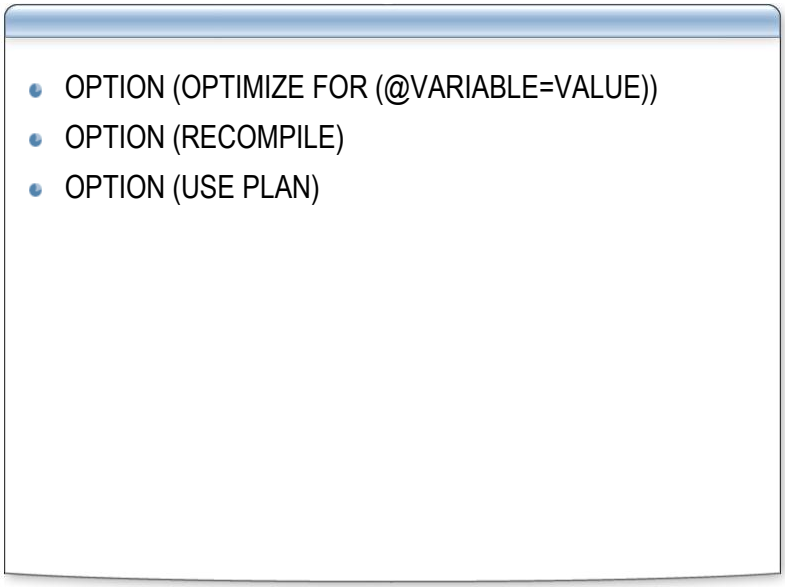
Query plans can be set directly by the use of query hints. Beginning with SQL 2005 options to optimize an execution of a stored procedure for certain parameter values were introduced. Plan guides allow for the plans to be predefined for queries.

## Objectives

After completing this section, you will be able to:

- Explain how plan guides influence the optimization of queries.
- Create a plan guide by using the OPTION (OPTIMIZE FOR) query hint.
- Optimize a query for a specific parameter value by using plan guides.
- Create a plan guide from cache in SQL Server 2008 to ensure a consistent query plan.

# Query Hints




## Issues relating to parameter sniffing

During the optimization of a stored procedure plan, if the value of a parameter is available, the query optimizer will ***sniff*** the incoming value and use this value for cardinality estimation. This is known as **parameter sniffing**, and in general, this results in a better execution plan.

Parameter sniffing can cause problems if the sniffed parameter value is not typical of the values that are actually used during a typical execution. For example, if a stored procedure is called the first time and the parameter passed returns 10 rows, the result is a query plan for that parameter. However, every subsequent call to the stored procedure passes in a parameter that returns 90 percent of the rows. The query plan based on the first execution will probably be inadequate for subsequent executions.

A second common scenario where parameter sniffing can cause problems is when the parameter is changed within the procedure before it is used in a query. Consider the following scenario:

```
Create procedure myproc @p1 int = null
As
Begin
       If @p1 is null
       Begin
              Select @p1 = 9999
       End
       Select … from T … WHERE c1 = @p1
End
```

If the procedure shown above passes NULL for the @p1 parameter, you can see that the procedure logic changes the value to 9999. The sniffed parameter value would be null, yet the runtime value would be 9999. Consequently, the cardinality estimates for the SELECT are inaccurate because the estimates were made using a value of NULL. To prevent this and other similar situations, you can use the following query options to specify how the query should be optimized:

### OPTION (OPTIMIZE FOR)

The **OPTION (OPTIMIZE FOR)** is most often used with a statement in a stored procedure to optimize a query for a certain value of a parameter that is used in the WHERE clause. It can also be used for SQL statements that are not in a stored procedure.

### OPTION (RECOMPILE)

This option instructs the query engine to generate a new plan each time the query is executed.

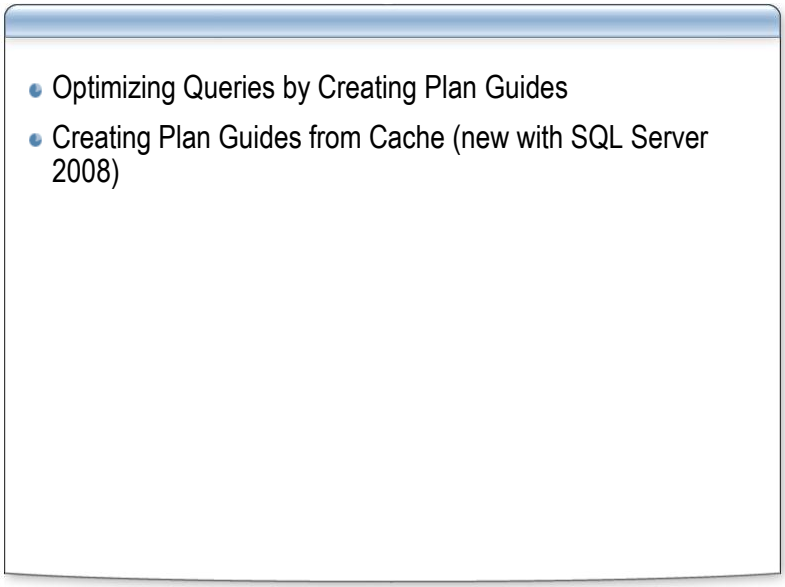
You should consider using OPTION (RECOMPILE) when:

- The plan is highly sensitive to the predicate value but there is no general ***good*** value to optimize for.
- The plan has range predicates that may vary significantly (for example, day, week, month, etc.)
- Only a single statement within a stored procedure needs to be recompiled. If the entire stored procedure needs to be recompiled on every execution, then you should consider using the **WITH RECOMPILE** clause. When a stored procedure is executed by using the **WITH RECOMPILE** clause, the entire stored procedure is recompiled. If you use the **OPTION (RECOMPILE)** query option, only the SELECT statement that uses the option needs to be recompiled.

### OPTION (USE PLAN)

This option is intended for scenarios where you do not have any control over the input SQL statements, for example when a third-party application is submitting ad hoc SQL statements. The USE PLAN option forces the optimizer to use a specified plan for the query.

# Optimizing queries by creating plan guides




Server 2005 introduces the sp_create_plan_guide system stored procedure for creating plan guides to optimize the performance of queries. You can use this procedure when you either cannot or do not want to change the text of the query directly. Plan guides can be useful when a small subset of queries in a database application deployed from a third-party vendor are not performing as expected. Plan guides influence the optimization of queries by attaching query hints to them. In the sp_create_plan_guide statement, you specify the query that you want optimized and the OPTION clause that contains the query hints that you want to use to optimize the query. When the query runs, SQL Server matches the query to the plan guide and attaches the OPTION clause to the query.

The following requirements apply to plan guides:

- Plan guides require either Standard or Enterprise Editions.
- Plan guides cannot be created against stored procedures, functions, or data manipulation language (DML) triggers that use **WITH ENCRYPTION** clause.
- If **@type = 'OBJECT'**, you need **ALTER** permission on the object. If **@type= 'SQL'** or **'TEMPLATE'**, you need **ALTER** permission on the database.
- When matching the text of a stored procedure or query against an existing plan guide, SQL Server performs case and accent sensitive matching even if the database is case insensitive. This does not apply to keywords. SQL Server Also ignores the carriage return, line feed, and space characters.

The syntax for the **sp_create_plan_guide** statement is:

```
sp_create_plan_guide [ @name = ] N'plan_guide_name'
  , [ @stmt = ] N'statement_text'
  , [ @type = ] N' { OBJECT | SQL | TEMPLATE }'
  , [ @module_or_batch = ]
      {
          N'[ schema_name.]object_name'
        | N'batch_text'
        | NULL
      }
  , [ @params = ] { N'@parameter_name data_type [,…n ]' | NULL }
  , [ @hints = ] { N'OPTION ( query_hint [,…n ] )' | NULL }
```

The following table describes the key parameters in the syntax for sp_create_plan_guide:

| Parameter | Description |
|---|---|
| @stmt | The statement whose plan you want to influence. |
| @type | The context in which statement appears.<br>• If **type** = **'OBJECT'**, the statement needs to appear in the context of a stored procedure, scalar function, multi-statement table-valued function, or T-SQL DML trigger.<br>• If **type** = **'SQL'**, the statement needs to appear as either a stand-alone statement or a statement in a batch.<br>• If **type** = '**TEMPLATE**', the plan guide applies to any query that parameterizes to the form indicated in the **@stmt** parameter. |
| @hints | Only OPTION query hints are allowed. If **TEMPLATE** is specified for the **type** parameter, the hint can be either **SIMPLE** or **FORCED**. |

The following example creates a plan guide:

```
Sp_create_plan_guide N'g1',
@stmt = N'select count(*) as c
   from Sales.SalesOrderHeader h, Sales.SalesOrderDetail d
   where h.SalesOrderID = d.SalesOrderID
and h.OrderDate between ''1/1/2000'' and ''1/1/2005''',
   @type = N'sql',
@module_or_batch = NULL,
@params = NULL,
@hints = N'option (merge join)'
```

If the query is issued as a stand-alone statement or a statement in a batch, the hint instructs the engine to use a merge join.

## Creating Plan Guides from Cache (new with SQL 2008)

Sometimes, a consistent plan is already cached, and it is beneficial to lock this plan in place so that it will not be affected by things such as a bulk insert of a large amount of sequential data. In this case, you can create a plan guide as was shown in the example in the previous topic, but there is a considerable amount of effort required to prepare and execute this. To simplify this, SQL 2008 introduced the **sp_create_plan_guide_from_handle** system stored procedure to lock an existing plan in place. This process is known as *freezing a query plan*.

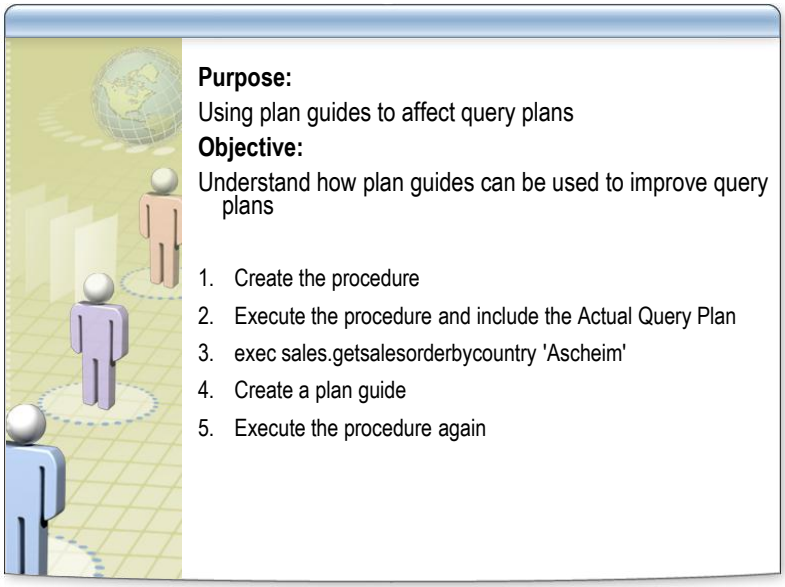
The syntax for the **sp_create_plan_guide_from_handle** statement is:

```
sp_create_plan_guide_from_handle [ @name = ] N'plan_guide_name'
    , [ @plan_handle = ] plan_handle
    , [ [ @statement_start_offset = ] { statement_start_offset | NULL } ]
```

The following table describes the key parameters in the syntax for sp_create_plan_guide_from_handle:

| Parameter | Description |
|---|---|
| @name | Specifies the name of the plan guide. |
| @plan_handle | Identifies a batch in the plan cache. plan_handle is varbinary(64), and it can be obtained from the sys.dm_exec_query_stats DMV. |
| @statement_start_offset | Identifies the starting position of the statement within the batch of the specified plan_handle. statement_start_offset is int, with a default value of NULL.<br>The statement *offset* corresponds to the statement_start_offset column in the sys.dm_exec_query_stats DMV. |

# Demonstration 5: Using Plan Guides



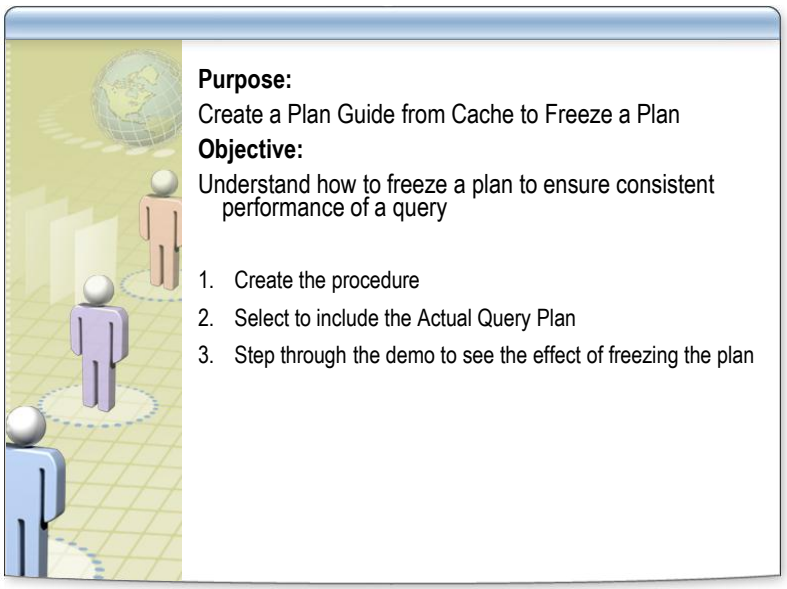

View the effect that the use of plan guides can have on query execution by comparing the execution plans with and without the plan guide in place. Run the following query:

```
CREATE PROCEDURE Sales.GetSalesOrderByCountry (@Country nvarchar(60))
AS
BEGIN
    SELECT *
    FROM Sales.SalesOrderHeader h, Sales.Customer c,
        Sales.SalesTerritory t
    WHERE h.CustomerID = c.CustomerID
        AND c.TerritoryID = t.TerritoryID
        AND CountryRegionCode = @Country
END
```

The following query creates the plan guide that can be used with the stored procedure created above:

```
sp_create_plan_guide
@name =  N'Guide1',
@stmt = N'SELECT *
        FROM Sales.SalesOrderHeader h,
        Sales.Customer c,
        Sales.SalesTerritory t
        WHERE h.CustomerID = c.CustomerID
            AND c.TerritoryID = t.TerritoryID
            AND CountryRegionCode = @Country',
@type = N'OBJECT',
@module_or_batch = N'Sales.GetSalesOrderByCountry',
@params = NULL,
@hints = N'OPTION (OPTIMIZE FOR (@Country = N''US''))‘
```

# Demonstration 6: Creating Plan Guides from Cache




Sometimes, parameter values can cause a large difference in query costs. This can also lead to performance problems depending on the parameter value with which the stored procedure was executed first. Freezing a plan allows you to gain consistency in these situations by guaranteeing that a known good query plan is used.

Run the following query to set up the demo:

```
use adventureworkspto
go

if exists (select 1 from sys.objects where type = 'P'
       and name = 'getRowsByStateProvinceID')
begin
       drop procedure getRowsByStateProvinceID
end
go

create procedure getRowsByStateProvinceID
@stateProvinceID int
as

select * from Person.Address a
       WHERE a.StateProvinceID = @stateProvinceID
go

dbcc freeproccache

set statistics time on
```

After the setup, ensure that the option to show the actual query plan is checked, and run the following query. On each execution, note the execution time in the **Messages** tab, and note the query plan in the **Query plan** tab of the Query window.

```
exec getRowsByStateProvinceID 119

exec getRowsByStateProvinceID 9
```

Clear the plan cache to see what happens if the stored procedure is executed with 9 first:

```
dbcc freeproccache

exec getRowsByStateProvinceID 9

exec getRowsByStateProvinceID 119
```

Note that the execution plan and the execution times are different if you run the stored procedure with 9 first. In this case, this plan is probably beneficial, because it does not have a potential to have problem with either order of execution. In this case, you should freeze this plan to ensure that you always get the same query plan. In order to do that, you need to find the plan_handle and statement_start_offset for this plan. You can locate plan_handle in your plan cache by using the following statement:

```
select s.plan_handle, s.statement_start_offset,
       t.text from sys.dm_exec_query_stats s
       cross apply sys.dm_exec_sql_text(s.sql_handle) t
```

Replace the plan_handle and statement_start_offset of the next query with the values retrieved from the previous query, and freeze the query plan by using the **create_plan_guide_from_handle** system stored procedure as follows:

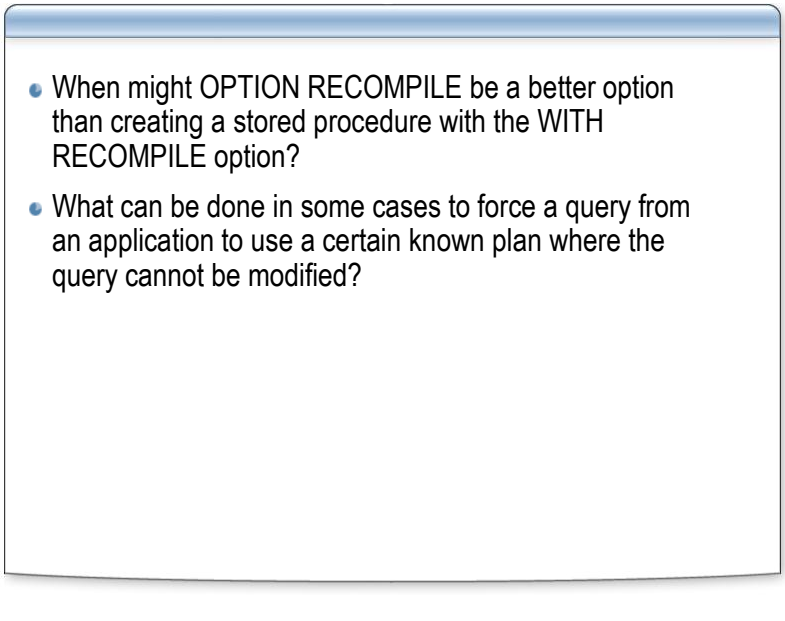
```
EXEC sp_create_plan_guide_from_handle @name = N'getAddresses'
,      @plan_handle = 0x050008003E6F336840E191840000000000000000000000000
,      @statement_start_offset = 146
```

Check the system catalogues to ensure that the plan guide is there, and then free the procedure cache as follows:

```
select * from sys.plan_guides

dbcc freeproccache
```

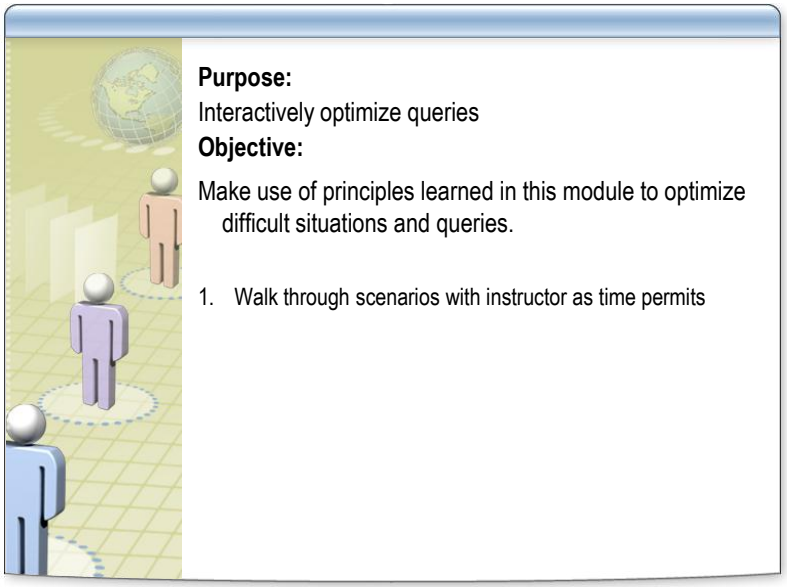
At this point, you should try clearing the procedure cache, and running the stored procedure with the different parameter values first. Now you can see that SQL Server always uses the query plan that joins before the key lookup. This is the plan that you froze. This will give you consistent performance in situations where parameter sniffing is a problem.

# Section 4 Review



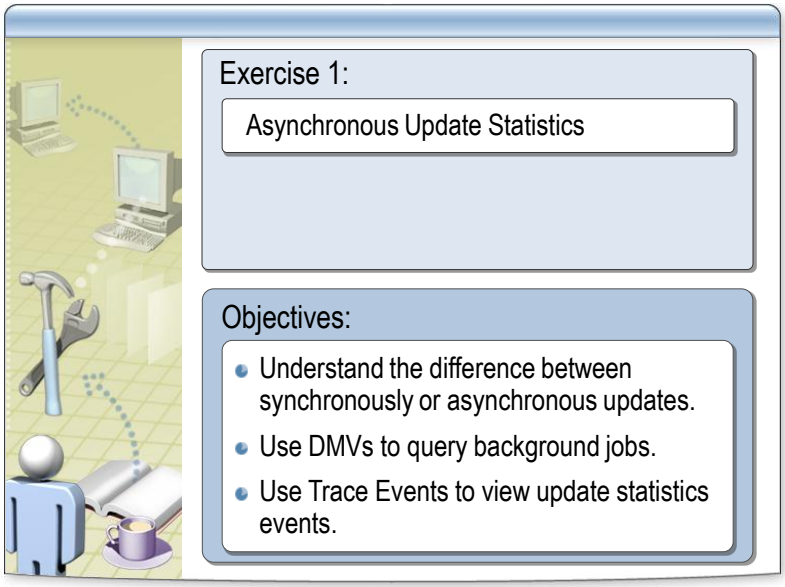

- When might OPTION RECOMPILE be a better option than creating a stored procedure with the WITH RECOMPILE option?
- What can be done in some cases to force a query from an application to use a certain known plan where the query cannot be modified?

# Demonstration 7, 8, and 9: Query Optimization (Optional)

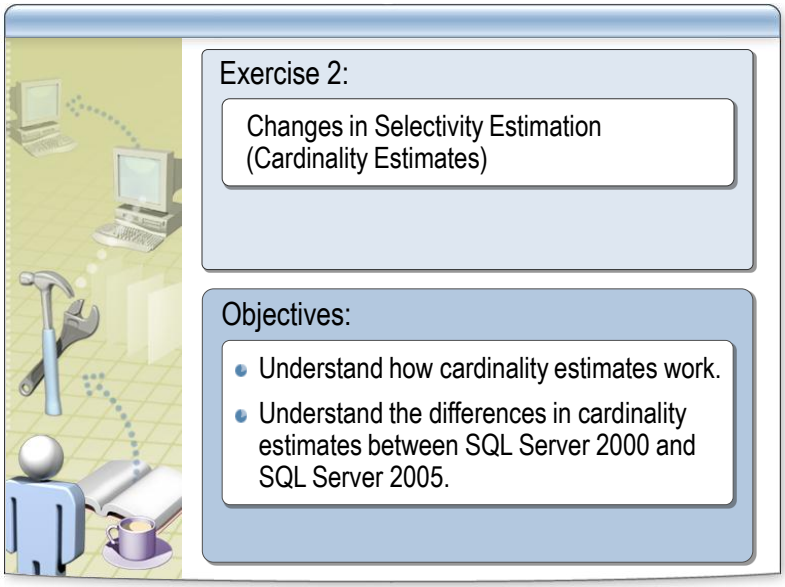



47

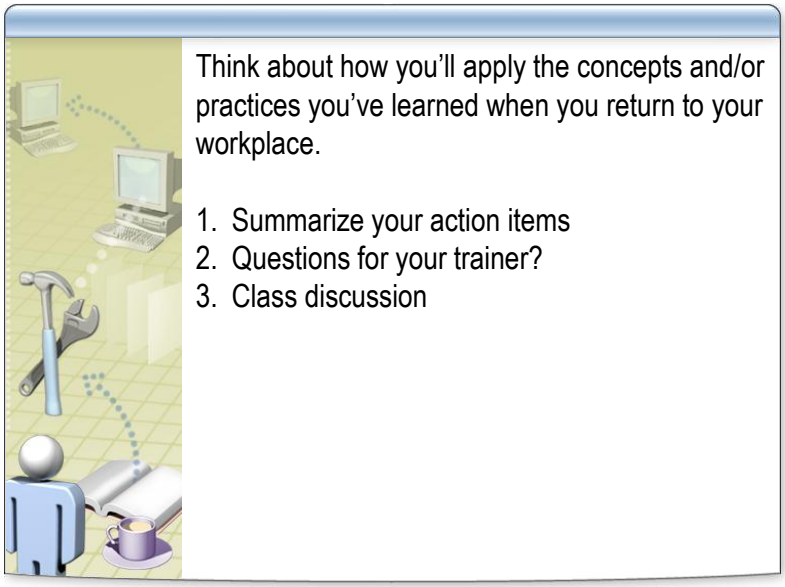
# Lab 1: Exercise 1



# Lab 1: Exercise 2



# Action Planning Exercise




When you return to your workplace, you'll want to use the concepts and practices you've learned to positively impact your IT environment. In this exercise you'll think about what you've learned and create action items that you can follow up on when you return to your workplace.

## Step 1: Summarize Your Action Items (5 Minutes)

How can you apply what you've learned to your workplace?

| | |
|---|---|
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |

## Step 2: Questions for your trainer? (5 Minutes)

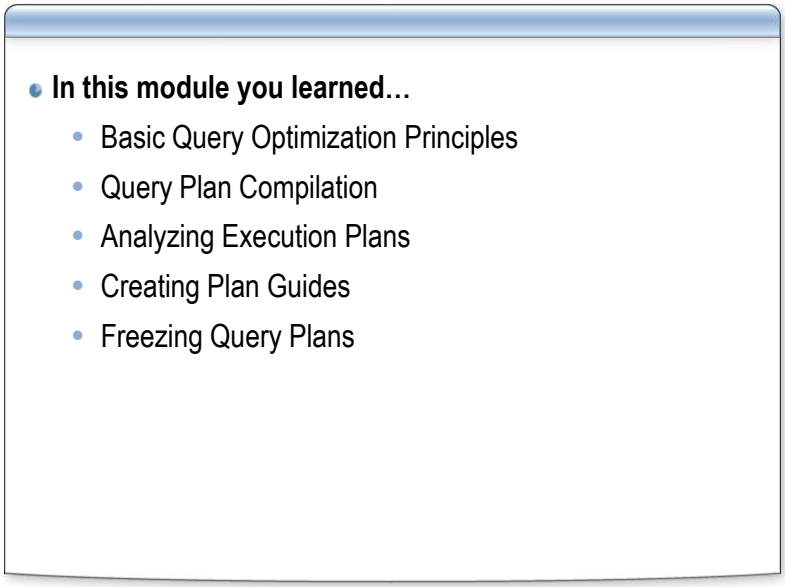
Record any questions you have about accomplishing the Action Items you listed above. The trainer will allow time for discussion before moving to the next module.

| | |
|---|---|
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |

## Step 3: Class Discussion

Notes:

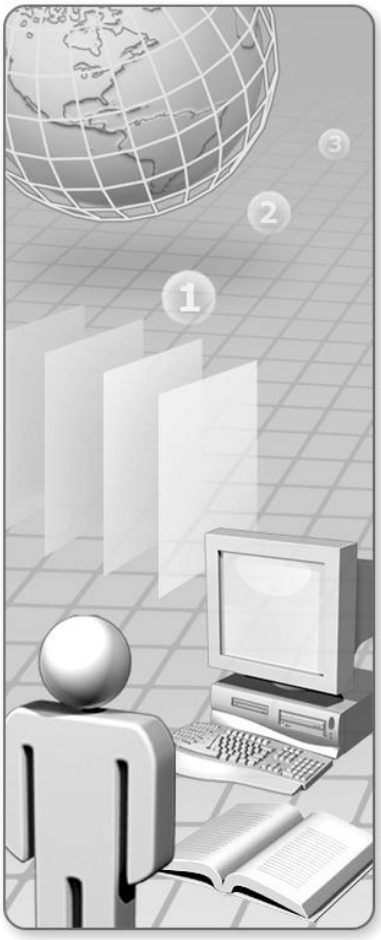
# Module Summary



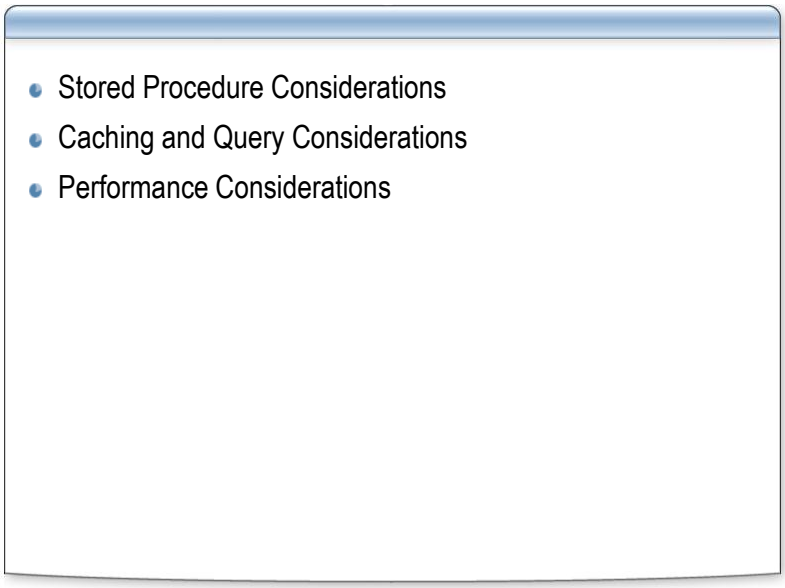

In this module, you learned about:

- Basic query optimization principles.
- Query plan compilation.
- Analyzing execution plans.
- Creating plan guides.
- Freezing query plans.

# Module 6: Programming Efficiency



# Module Overview




## Introduction

Programming SQL Server efficiently means more than just writing efficient queries. Mechanisms such as stored procedures offer ways to store complex logic on SQL Server and call them consistently. Beginning with SQL Server 2005, stored procedures can be written in Common Language Runtime (CLR) languages as well.

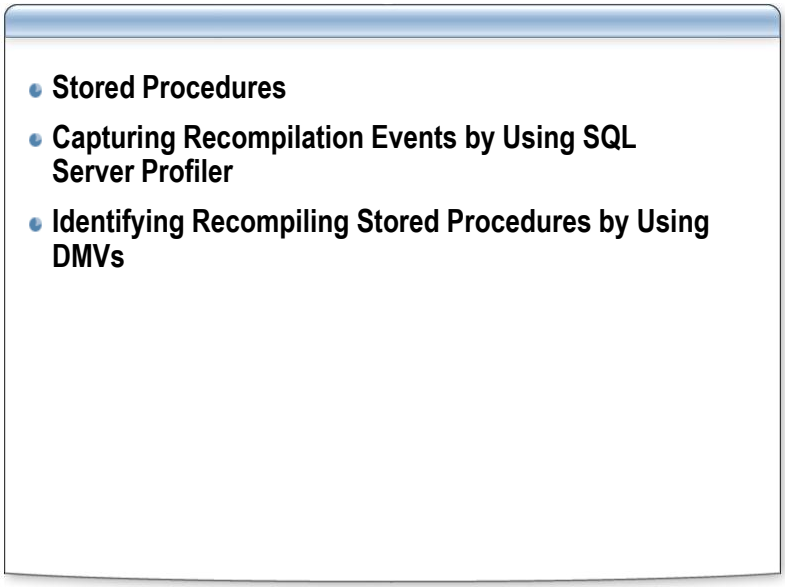
This module explains how to write efficient stored procedures, and efficiently use the SQL Server caching mechanisms to ensure consistent performance, without the need for re-compilations.

## Objectives

After completing this module, you will be able to:

- Optimize the storage and compilation of stored procedures.
- Optimize queries and caching.
- Design databases for performance.

# Section 1: Stored Procedure Considerations




## Introduction

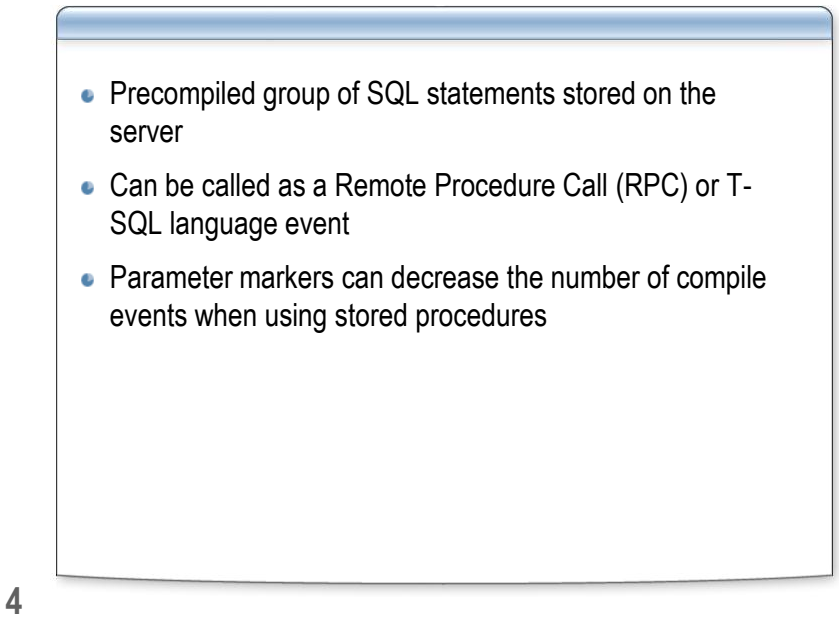
Stored procedures offer a way to store batches of T-SQL code centrally on the server. This is convenient and efficient, but requires proper consideration. In this section you will explore how to maintain efficiency by using stored procedures.

## Objectives

After completing this section, you will be able to:

- Explain how SQL Server stores, compiles, and optimizes stored procedures.
- Capture recompilation events by using SQL Server Profiler.
- Return information about stored procedure recompilation by using dynamic management views (DMVs).

# Calling Stored Procedures




SQL Server provides the ability to precompile a group of Transact-SQL (T-SQL) statements and store these statements on the server. These specialized T-SQL scripts are called ***stored procedures***. The execution plans for these stored procedures are reusable, and stay cached and available for subsequent use. The query optimizer may use significant amounts of CPU resources to evaluate thousands of possible execution plans in the process of compilation and optimization. Therefore, reusing a saved plan can provide you with a major performance advantage.

The SQL statements that you use to create the stored procedure are stored in the **sys.sql_modules** system table. The first time a stored procedure is called after SQL Server restarts, the stored procedure text is retrieved from sys.sql_modules and an execution plan is compiled and cached. This cached execution plan is then available for reuse on subsequent calls to the stored procedure.

## Calling stored procedures

You can call stored procedures either as T-SQL language events or by using the remote Procedure Call (RPC) protocol.

### T-SQL language events

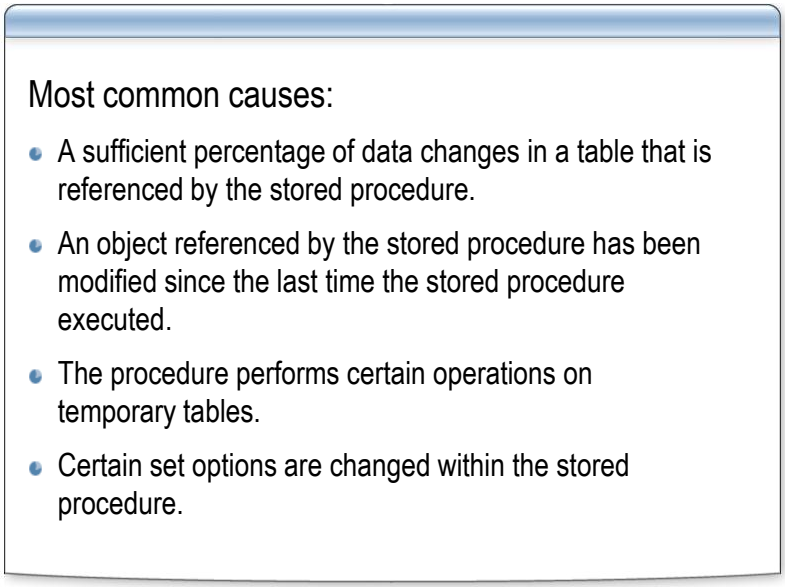
Stored procedures are called as T-SQL language events either when the application uses syntax such as the T-SQL "EXEC" or when the application does not specifically submit the query with a type of *Stored procedure*. When a stored procedure is called as a T-SQL language event, SQL must first parse and compile the query, check that the query is not a

SELECT, INSERT, UPDATE, DELETE or data definition language (DDL) statement, decide that the query is trying to run a stored procedure, and then try to find a plan in cache for that procedure.

### RPC

To avoid the situation where SQL must first parse and compile the language event, you can submit the call by using the RPC protocol. When SQL Server receives a call via RPC, it knows that it is receiving a call to a stored procedure. Therefore, it is able to bypass most of the additional preparatory work required to run the stored procedure. However; this optimization also bypasses most of the parameter checking. For this reason, you must ensure that your application has appropriate checks to prevent invalid input.

# Stored Procedure Recompilation




Beginning with SQL Server 2005, the recompilation of stored procedures is performed on individual statements within the stored procedure rather than on the entire stored procedure. Although recompiles are less resource intensive than recompiling the entire stored procedure, unnecessary recompiles still consume CPU resources. In cases where it becomes a problem, it may be necessary to troubleshoot the recompilation of stored procedures.

Here are a few common causes of recompilation:

- If a sufficient percentage of data has changed in a table referenced by a stored procedure, since the time the original query plan was generated, SQL Server will recompile the stored procedure to ensure that it has a plan based on the most up-to-date statistical data. The algorithm that SQL Server uses to determine whether a plan should be recompiled is the same as the algorithm that it uses for auto-update statistics.

  If auto-update statistics is enabled, an **auto stats** event in Profiler can help confirm that the recompile was due to the number of row modifications.

- A stored procedure is compiled the first time it runs. If an object is modified, or dropped and recreated between executions, then when each statement within a stored procedure that references this object runs, the statement is recompiled. Although statement-level recompiles should be much less expensive than stored procedure recompiles, SQL still must take compile locks in order to complete the action. If many recompiles are generated by execution of a stored procedure that is heavily

used on a system, then it is possible that this will still drive up CPU utilization and/or create a blocking situation.

To minimize this risk, you should carefully evaluate the need for DDL in stored procedures. You should check if the operations in the stored procedure can be most efficiently performed by the use of temporary tables, or by using other DDL statements. In that case, you should carefully evaluate the placement of the DDL to get the best performance of the data manipulation language (DML) statements in the stored procedure while minimizing the number of statements that must be recompiled after the execution of the DDL.

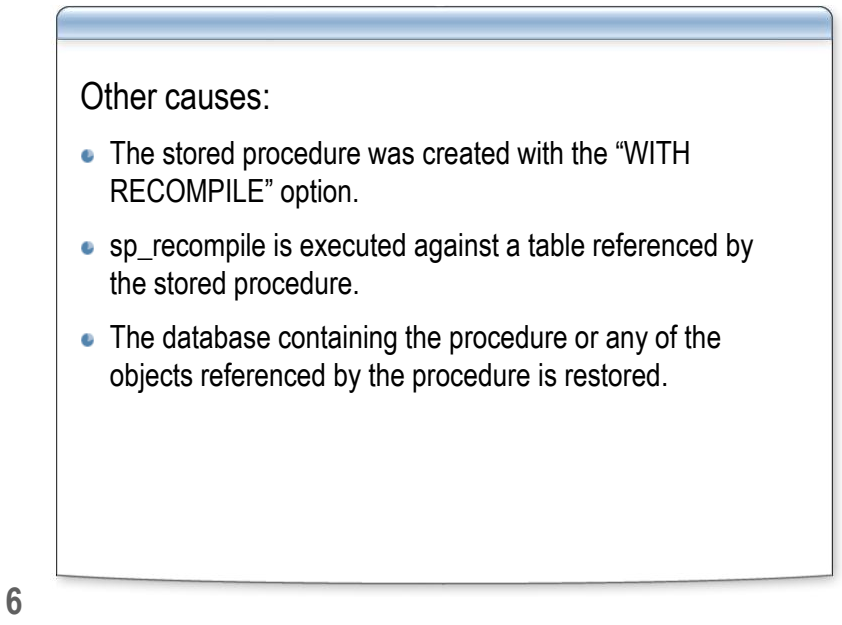
**Note**: For more information, refer to the topic *Description of SQL Server blocking caused by compile locks* at http://support.microsoft.com/?id=263889.

- Use of temporary tables in a stored procedure may cause the stored procedure to be recompiled every time the procedure is executed. This may be inconsequential in large batches, but in Online Transaction Processing (OLTP), this may be an unnecessary overhead.
- The following five SET options are set to ON by default:
  - ANSI_DEFAULTS
  - ANSI_NULLS
  - ANSI_PADDING
  - ANSI_WARNINGS
  - CONCAT_NULL_YIELDS_NULL

  If you run the statement set operation to set any of these options to OFF, the statements within the stored procedure affected by the changes will be recompiled the first time the procedure runs. The reason for this is that changing these options may affect the query result that triggered the recompilation.

**Note**: For more information on identifying the causes of recompilations in SQL Server 2005, refer to *How to identify the cause of recompilation in an SP:Recompile event* at http://support.microsoft.com/kb/308737/.

# Stored Procedure Recompilation (continued)



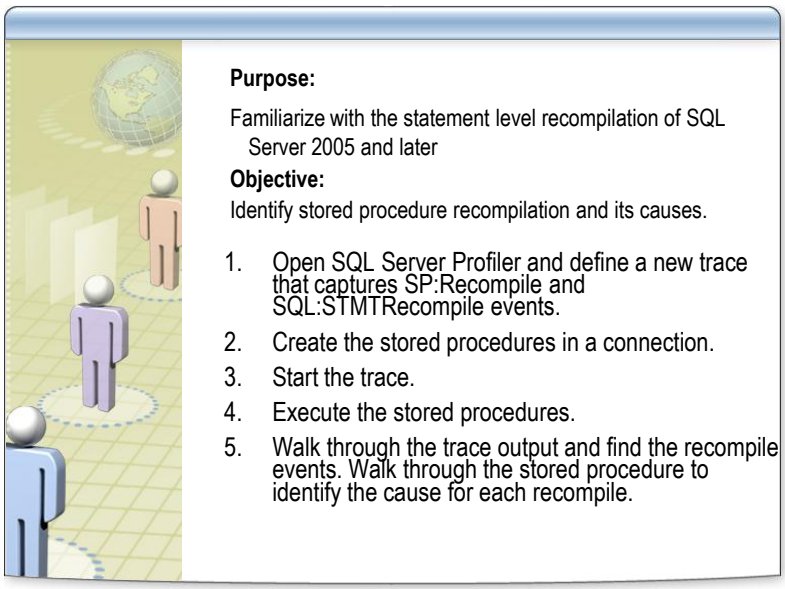

- Creating a stored procedure that specifies the WITH RECOMPILE option in its definition indicates that SQL Server does not cache a plan for this stored procedure; the stored procedure is recompiled each time it is executed. You should use the WITH RECOMPILE option when stored procedures take parameters whose values differ widely between executions of the stored procedure, thereby resulting in different execution plans to be created each time. Use of this option is uncommon and using it imporperly causes the stored procedure to run more slowly, because the stored procedure must be recompiled each time it is executed.

  If you want only individual queries inside the stored procedure to be recompiled, rather than the entire stored procedure, you should specify the RECOMPILE query hint inside each query that you want recompiled. This behavior mimics the statement-level recompilation behavior in SQL Server noted above, but in addition to using the current parameter values of the stored procedure, the RECOMPILE query hint also uses the values of any local variables inside the stored procedure when compiling the statement. You should use this option when atypical or temporary values are used in only a subset of queries belonging to the stored procedure.

- The **sp_recompile** system stored procedure forces a recompile of a stored procedure the next time it is run. Additionally, running sp_recompile on a table or view that is referenced by a stored procedure will cause that stored procedure to recompile the next time it is run.
- If a database containing a stored procedure is restored, then that stored procedure will recompile on its next execution. If a stored procedure references objects in another

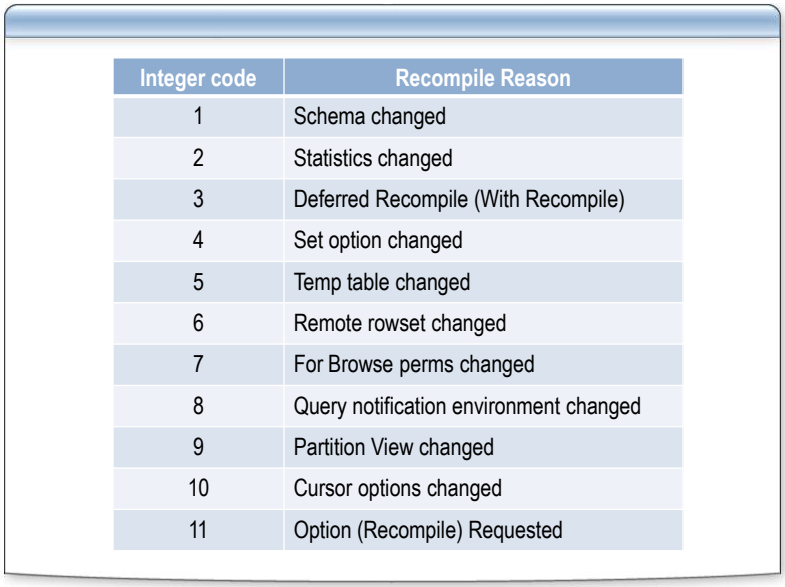
database, and that database is restored, then the statements within the referencing stored procedure which access these objects will be recompiled on the next execution.

# Demonstration 1: Recompilation




7

# Capturing Recompilation Events by Using SQL Server Profiler



| Integer code | Recompile Reason |
|---|---|
| 1 | Schema changed |
| 2 | Statistics changed |
| 3 | Deferred Recompile (With Recompile) |
| 4 | Set option changed |
| 5 | Temp table changed |
| 6 | Remote rowset changed |
| 7 | For Browse perms changed |
| 8 | Query notification environment changed |
| 9 | Partition View changed |
| 10 | Cursor options changed |
| 11 | Option (Recompile) Requested |

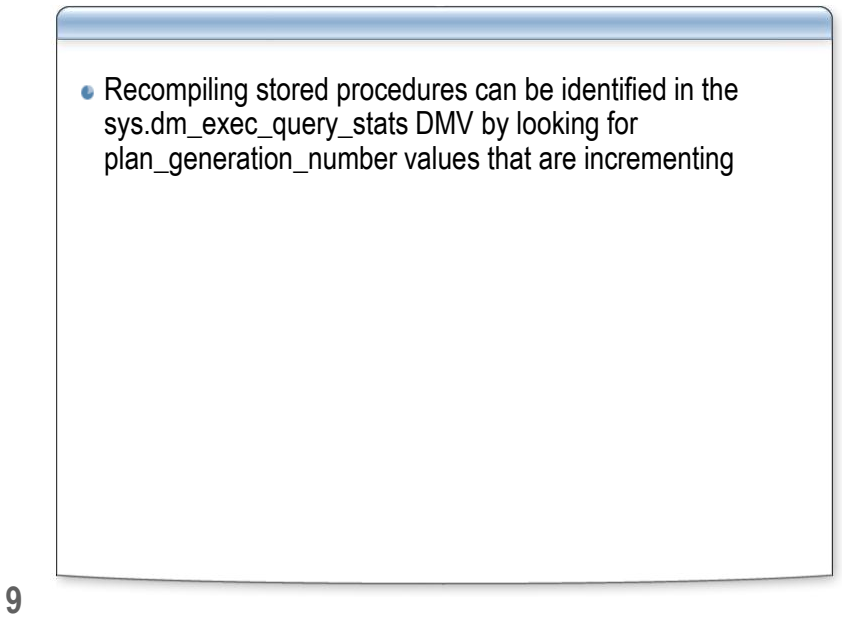
8

When SQL Server runs a stored procedure or a trigger, it may need to recompile the stored procedure or the trigger for various reasons.

You can capture recompile events in traces by capturing the T-SQL **SQL:StmtRecompile** event in the SQL Server Profiler. This event indicates that SQL Server recompiled a statement. The ***Event Subclass*** column of this trace event provides the information necessary to pinpoint the cause of the recompile. The slide above shows the SQL Server 2005 integer codes with the reason for recompilation.

> **Note:** For more information, refer to the topic *How to identify the cause of recompilation in an SP:Recompile event* at http://support.microsoft.com/?kbid=308737.

# Identifying Recompiling Stored Procedures by Using DMVs




SQL traces provide the most details about recompiles and their causes, but you must set up traces and run them at the time of the recompile event to capture any information. In cases where traces are not running, you can determine the time of the last recompile on each stored procedure or statement by querying dynamic management views (DMVs).

The following query returns information that you can use to identify issues with stored procedure or trigger recompilation:

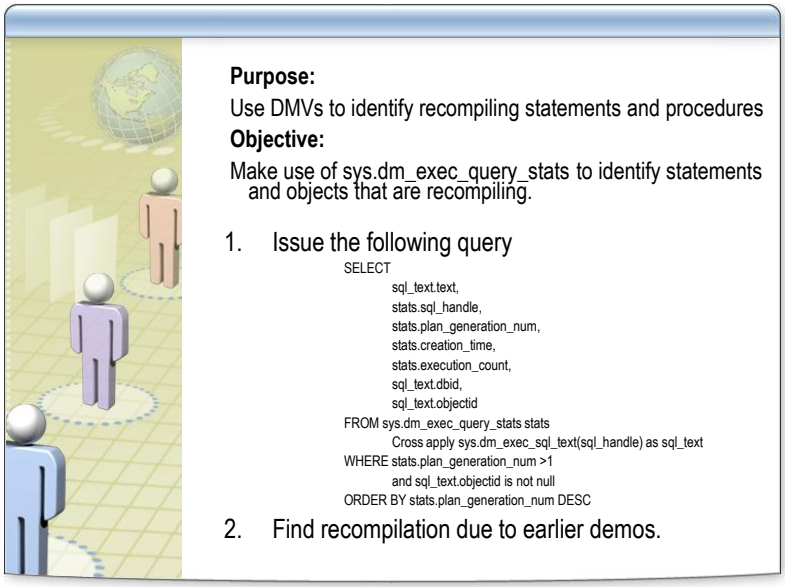
```
select
       sql_text.text,
       stats.sql_handle,
       stats.plan_generation_num,
       stats.creation_time,
       stats.execution_count,
       sql_text.dbid,
       sql_text.objectid
from sys.dm_exec_query_stats stats
       Cross apply sys.dm_exec_sql_text(sql_handle) as sql_text
where stats.plan_generation_num >1
       and sql_text.objectid is not null
order by stats.plan_generation_num desc
```

The query above returns the following information:

| Column | Description |
|---|---|
| sql_text.text | Contains the text of the stored procedure or the trigger that is being recompiled. |

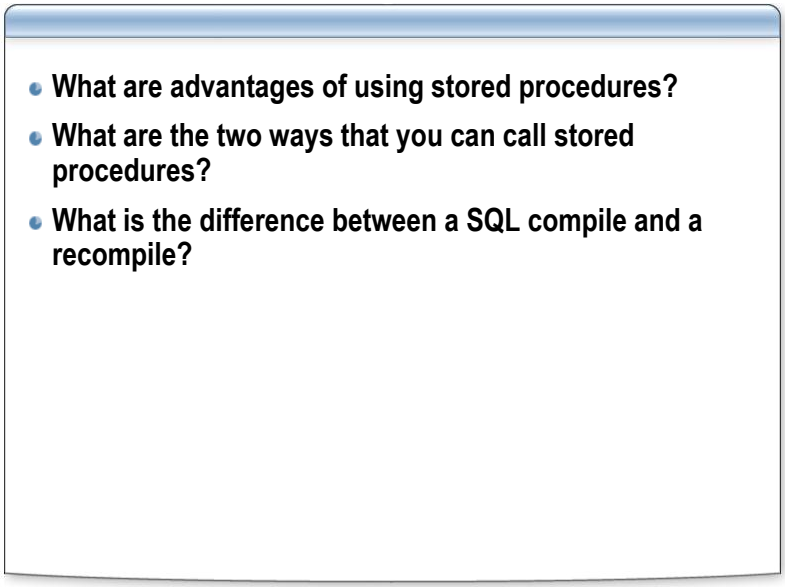
| Column | Description |
|---|---|
| stats.plan_generation_num | Contains the number of times that this procedure has been recompiled while in the cache. Limiting the query to only those rows where this number is greater than 1 ensures that the query only returns stored procedures that have been recompiled. By ordering this column in descending order, the query lists recompiled stored procedures at the top of the list, enabling you to focus on the stored procedures with the most recompiles when resolving recompile issues. |
| stats.creation_time | Contains the time when the plan was compiled. |
| stats.execution_count | Contains the number of times this procedure or trigger has executed since it was last compiled. |
| sql_text.dbid and sql_text.objectid | Contain the name of the stored procedure or trigger for which this plan exists. The query filters out the rows where **objectid** is null to filter out all ad hoc queries. This ensures that the query returns only those plans for which there is a stored procedure or trigger defined as an object within the database. |

# Demonstration 2: Identifying Recompiling Statements



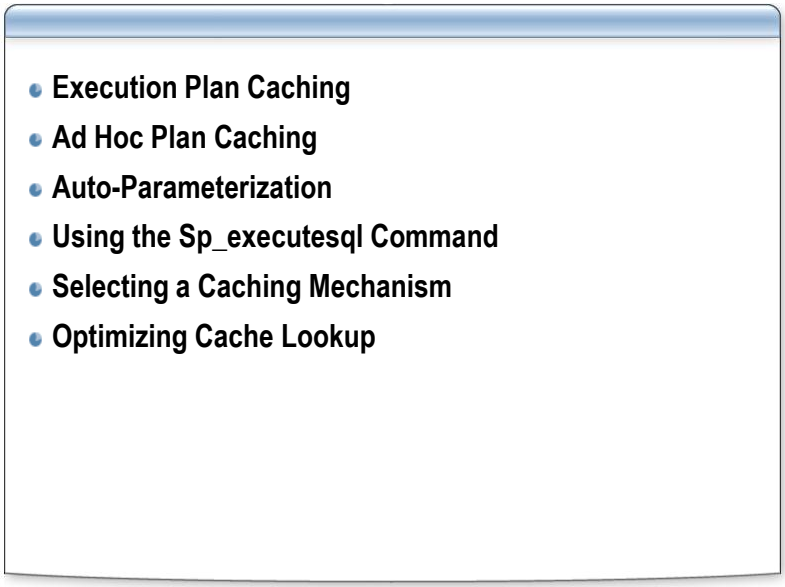

10

# Section 1 Review




- What are advantages of using stored procedures?
- What are the two ways that you can call stored procedures?
- What is the difference between a SQL compile and a recompile?

# Section 2: Caching and Query Considerations




## Introduction

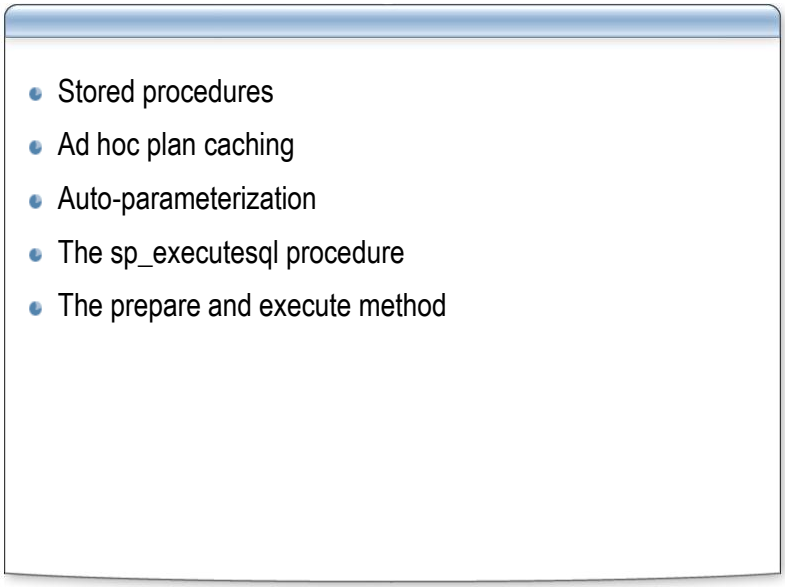
SQL Server provides several distinct methods of caching and reusing query execution plans. Which mechanism you should use depends on the specific situation. You should understand these mechanisms so that you can effectively choose the right mechanism for your situation.

## Objectives

After completing this section, you will be able to:

- List the mechanisms used by SQL for execution plan caching.
- Identify the situations where SQL Server uses ad hoc execution plan caching.
- Differentiate between simple and forced parameterization.
- Explain the benefits of using the sp_executesql command.
- Select a caching mechanism for a given set of requirements.

# Execution Plan Caching

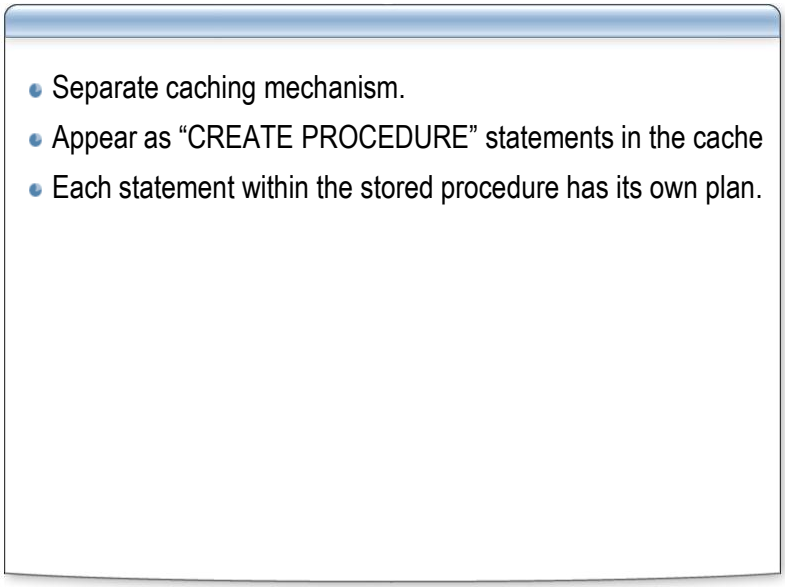



You can reduce SQL Server recompilation time by using one of the following five mechanisms available in SQL Server for execution plan caching:

- Stored procedures
- Ad hoc caching
- Auto-parameterization
- The **sp_executesql** stored procedure
- The prepare and execute method

Each mechanism is described in detail in the subsequent topics.

# Stored Procedure Caching




Stored procedures come with their own caching mechanism in SQL Server. Consequently, when querying the procedure cache, stored procedures will appear with a look much different than what is visible with plans using other caching mechanisms.

Open the Query window and run the following:

```
/* run to End of setup comment */

use adventureworkspto;

-- free the procedure cache to get rid of all the noise:

DBCC FREEPROCCACHE;
go
/* End of setup */

-- execute the following system stored procedure to get the plan in cache:

exec sp_helptext uspGetBillOfMaterials
```

The above query will clear the noise out of the cache, and run the system stored procedure **sp_helptext** so that you will get the query plan for that stored procedure cached.

Now, run the following query to look at the plan cache:

```
SELECT s.sql_handle, s.statement_start_offset, s.statement_end_offset
, s.creation_time, s.last_execution_time, t.text
 FROM sys.dm_exec_query_stats s
       CROSS APPLY sys.dm_exec_sql_text(s.sql_handle) t
```

You will notice that several lines appear which begin with **create procedure sys.sp_helptext**. This is not a statement creating the stored procedure, but rather reflects the way that the text appears for a cached stored procedure. The appearance of ***create procedure*** in the **text** column of **sys.dm_exec_sql_text** just indicates that stored procedures have executed and their execution plans are cached.

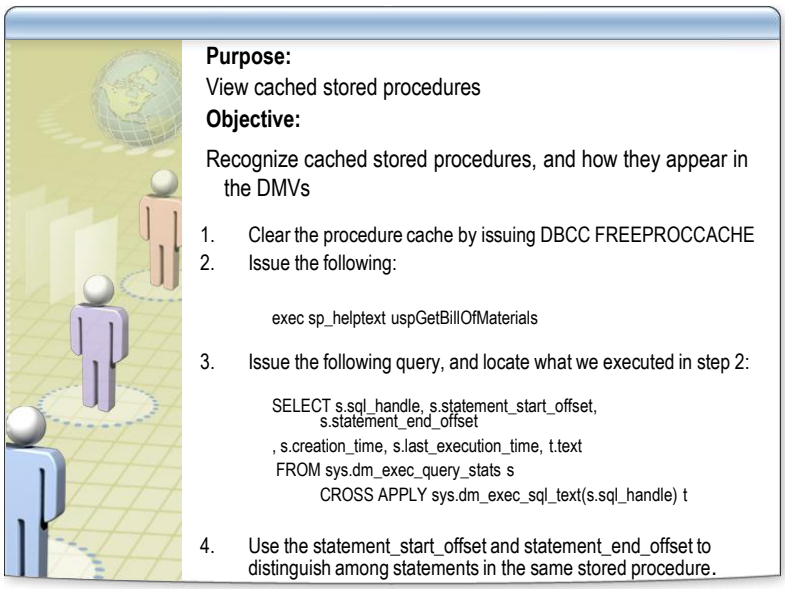
You will notice that although the entries in the **text** column are the same for all of the sp_helptext entries, each entry has a different **statement_start_offset** and **statement_end_offset**. Each line is the entry for a different statement within the stored procedure. Try this:

```
SELECT s.sql_handle, s.statement_start_offset, s.statement_end_offset
, s.creation_time, s.last_execution_time, t.text,
[statement] = substring(t.text, statement_start_offset/2,
       case statement_end_offset when -1 then datalength(t.text)
              else (statement_end_offset - statement_start_offset) / 2 end)
 FROM sys.dm_exec_query_stats s
       CROSS APPLY sys.dm_exec_sql_text(s.sql_handle) t
```

Now you can see each statement within the stored procedure in the **statement** column.

The mechanism used for caching stored procedures will not cause a separate entry to be created if the stored procedure is called again with characters in a different case. This differs from some other mechanisms. Having each statement with its own cached plan and its own entry allows for the statement-level recompile of stored procedures in SQL Server 2005 and later.

# Demonstration 3: Find Stored Procedures in Cache



**Purpose:**

View cached stored procedures

**Objective:**

Recognize cached stored procedures, and how they appear in the DMVs

1. Clear the procedure cache by issuing DBCC FREEPROCCACHE
2. Issue the following:

```
exec sp_helptext uspGetBillOfMaterials
```

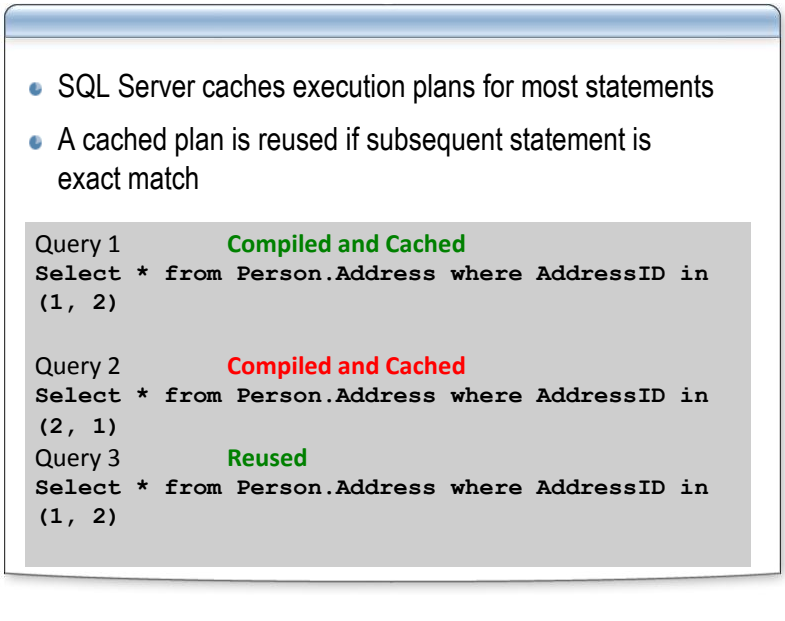
3. Issue the following query, and locate what we executed in step 2:

```
SELECT s.sql_handle, s.statement_start_offset,
        s.statement_end_offset
, s.creation_time, s.last_execution_time, t.text
 FROM sys.dm_exec_query_stats s
        CROSS APPLY sys.dm_exec_sql_text(s.sql_handle) t
```

4. Use the statement_start_offset and statement_end_offset to distinguish among statements in the same stored procedure.

15

# Ad Hoc Plan Caching




SQL Server caches the execution plans from ad hoc queries, and if a subsequent statement matches exactly, it uses the cached plan. This feature does not require any extra work, but it is limited to exact textual matches.

By default, SQL Server does not cache zero-cost plans.

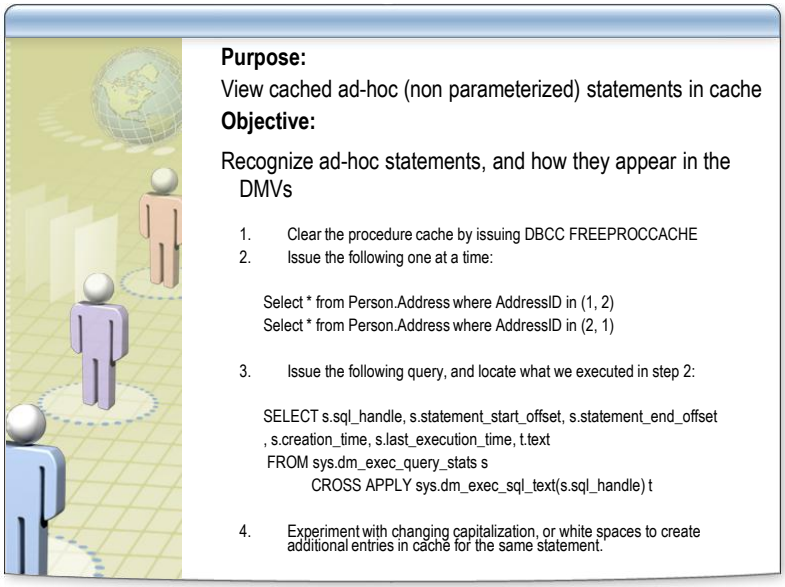
Run each of these statements individually.

```
Select * from Person.Address where AddressID in (1, 2)

Select * from Person.Address where AddressID in (2, 1)

Select * from Person.Address where AddressID in (1, 2)
```

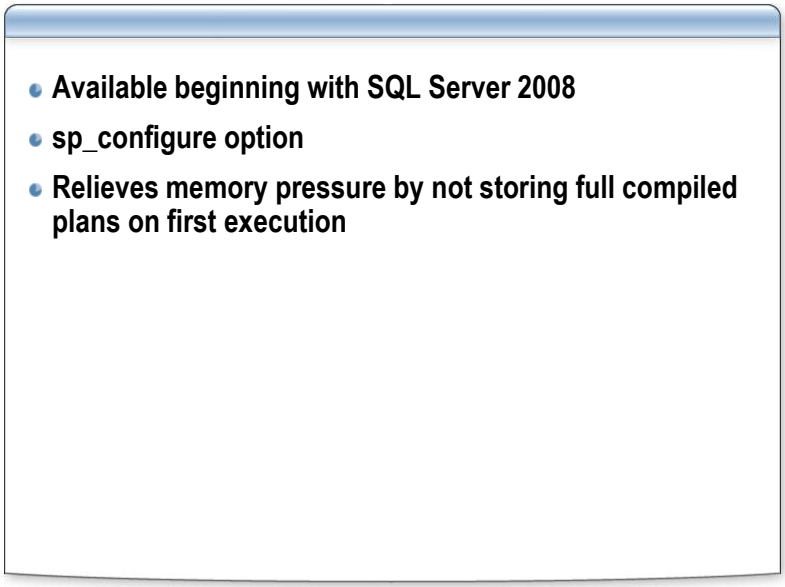
Query the cache again with the following query to see that SQL will not use the plans interchangeably:

```
SELECT s.sql_handle, s.statement_start_offset, s.statement_end_offset
, s.creation_time, s.last_execution_time, t.text
 FROM sys.dm_exec_query_stats s
       CROSS APPLY sys.dm_exec_sql_text(s.sql_handle) t
```

# Demonstration 4: Identify Ad-hoc Statements in Cache



# Optimizing for ad hoc workloads




In SQL Server 2008, an option was added to optimize SQL Server caching for ad hoc workloads. This new option can be viewed with the sp_configure options. It is the **optimize for ad hoc workloads** option.

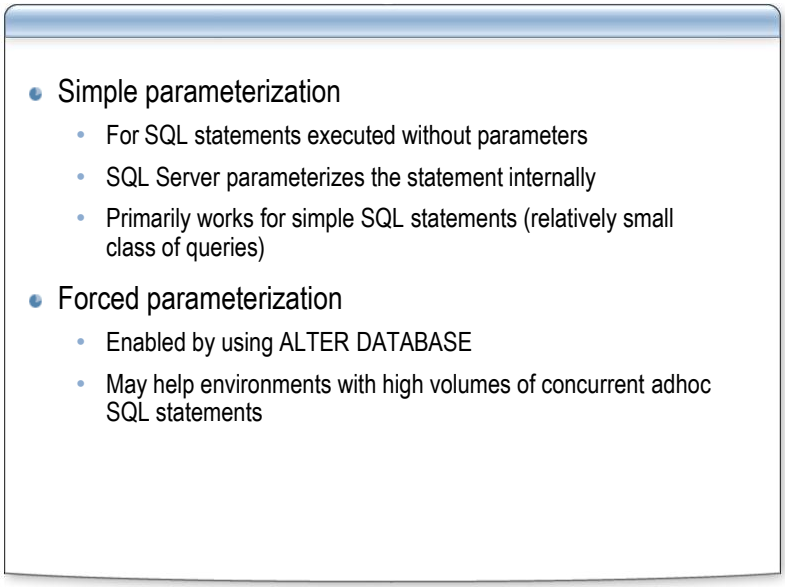
The **optimize for ad hoc workloads** option is used to improve the efficiency of the plan cache for workloads that contain many single use ad hoc batches. When this option is set to 1, the Database Engine stores a small compiled plan stub in the plan cache when a batch is compiled for the first time, instead of the full compiled plan. This helps to relieve memory pressure by not allowing the plan cache to become filled with compiled plans that are not reused.

The compiled plan stub allows the Database Engine to recognize that this ad hoc batch has been compiled before but has only stored a compiled plan stub, so when this batch is invoked (compiled or executed) again, the Database Engine compiles the batch, removes the compiled plan stub from the plan cache, and adds the full compiled plan to the plan cache.

Setting the optimize for ad hoc workloads to 1 affects only new plans; plans that are already in the plan cache are unaffected.

The compiled plan stub is one of the **cacheobjtypes** displayed by the **sys.dm_exec_cached_plans** catalog view. It has a unique sql handle and plan handle. The compiled plan stub does not have an execution plan associated with it and querying for the plan handle will not return an XML Showplan.

# Auto-Parameterization




## Simple parameterization

In SQL Server, using parameters or parameter markers in T-SQL statements increases the ability of the relational engine to match new SQL statements with existing, unused execution plans.

> **Note**: Parameter markers appear in the query as question marks in place of values that need to be added at the time of execution.

If a SQL statement is executed without parameters, SQL Server attempts to parameterize the statement internally to increase the possibility of matching it against an existing execution plan. Consider the following statement:

```
select * from Adventureworks.Person.Address where AddressID = 1
```

The value 1 at the end of the statement above can be specified as a parameter. The relational engine builds the execution plan for this batch as if a parameter had been specified in place of the value 1. Because of this auto-parameterization, SQL Server recognizes that the following two statements generate essentially the same execution plan and reuses the first plan for the second statement:

```
select * from Adventureworks.Person.Address where AddressID = 1
select * from Adventureworks.Person.Address where AddressID = 2
After running the queries shown above on your SQL Server 2005 database server, run
the following query to verify that auto-parameterization has taken place:select
sql_text.text, stats.execution_count
       from sys.dm_exec_query_stats stats
```

```
cross apply sys.dm_exec_sql_text(sql_handle) sql_text
```

If the procedure cache on your test machine is large, you may want to either clear it first or filter the results by adding a WHERE clause to return only those results where the text contains **Person**. Adding the WHERE clause limits the result enough so that you can easily find the queries that you ran in this demonstration.

A parameterized query should now appear in the procedure cache for the query of Address. The query should look like this:

```
 (@1 tinyint)SELECT * FROM [Adventureworks].[Person].[Address] WHERE
[AddressID]=@1
```

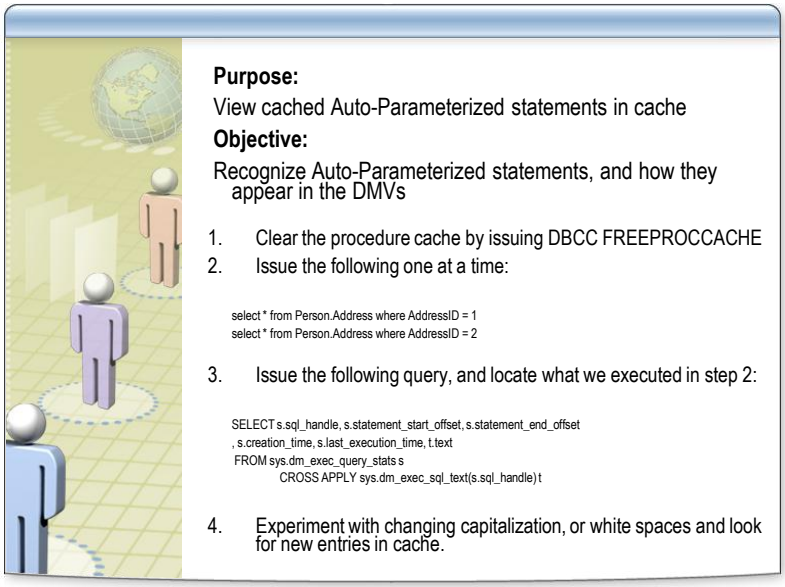
## Forced parameterization

When you set the PARAMETERIZATION option to FORCED, it overrides the default simple parameterization behavior for SQL Server. Forced parameterization causes any literal value that appears in a SELECT, INSERT, UPDATE, or DELETE statement to convert to a parameter during query compilation.

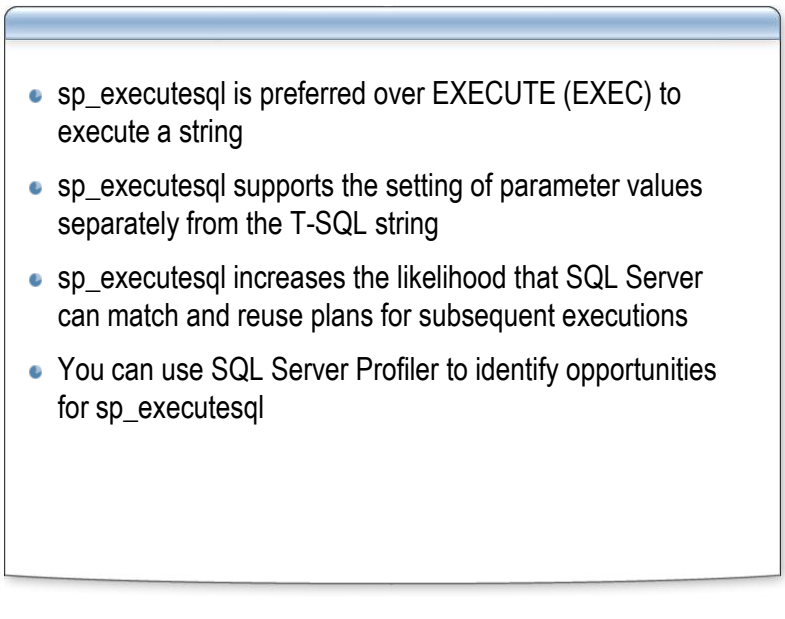
> **Note: Some queries** cannot be parameterized. For a list of exceptions, refer to the topic *Forced Parameterization* in SQL Server Books Online.

Forced parameterization might improve performance for environments where high volumes of concurrent ad hoc SQL statements are submitted. However; forced parameterization builds a query plan based on the parameter value the first time the query is called. This may result in suboptimal plans in subsequent executions.

# Demonstration 5: Identify Auto-Parameterized Statements in Cache



# The sp_executesql Stored Procedure




T-SQL supports two methods of building SQL statements at run time in T-SQL scripts, stored procedures, and triggers. These are:

- Using the **sp_executesql** system stored procedure to run a Unicode string. The sp_executesql stored procedure supports parameter substitution similar to the RAISERROR statement.
- Using the EXECUTE statement to run a character string. The EXECUTE statement does not support parameter substitution in the executed string.

When you use sp_executesql with parameters, it increases the chance that a query plan is reusable, and can be used to create reusable plans on queries that cannot be auto parameterized. Consider the following query:

```
select * from Adventureworks.Person.Address where AddressID in (1, 2)
```

The discussion of auto-parameterization in the previous topic used a similar query, except that the WHERE clause specified **AddressID** = 1. SQL Server is able to auto-parameterize that query. However, SQL Server will not auto-parameterize this new query when parameterization is set to simple because it uses the IN clause. If you query **sys.db_exec_query_stats** and cross apply the **sys.dm_exec_sql_text** dynamic management function (DMF), it shows that this new query has been cached literally, without the use of parameters. Because of this, SQL Server can only reuse this plan if the exact same query is issued by using the exact same values in the exact same order in the IN clause.

In this case, you can still create a reusable plan by using the **sp_executesql** system stored procedure, as follows:

```
exec sp_executesql
       N'select * from Adventureworks.Person.Address where AddressID in (@1, @2)'
       , N'@1 int, @2 int'    -- declare all variables used in the statement
       , 1, 2                 -- list the values for the variables
                              -- in the order they are used
```

If you query **sys.dm_exec_sql_text** again after running the query above, you will see that the following parameterized query plan has been created:

```
(@1 int, @2 int)select * from Adventureworks.Person.Address where AddressID in (@1,
@2)
```

If you run the same query with the values 3 and 4 instead of 1 and 2, SQL Server will not need to compile a new execution plan. Instead, it will find the existing plan and reuse it to return results where the AddressID is 3 or 4.

### Using sp_execute from an application

Applications can use parameter markers (the question marks) to optimize the reuse of a T-SQL statement with different input and output values. Consider the following query:

```
SELECT AddressLine1, AddressLine2 FROM Person.Address
WHERE Addressid = ?
```

If the application uses parameter markers with calls to SQLExecDirect (for Open Database Connectivity (ODBC)) or ICommandText::Execute (for object linking and embedding database (OLE DB)), the driver or provider automatically packages the SQL statement and runs it as an sp_executesql call. The statement does not need to be prepared and executed separately. When SQL Server receives a call to sp_executesql, it automatically checks the procedure cache for a matching plan and reuses that plan if it exists, or generates a new plan if no match is found.

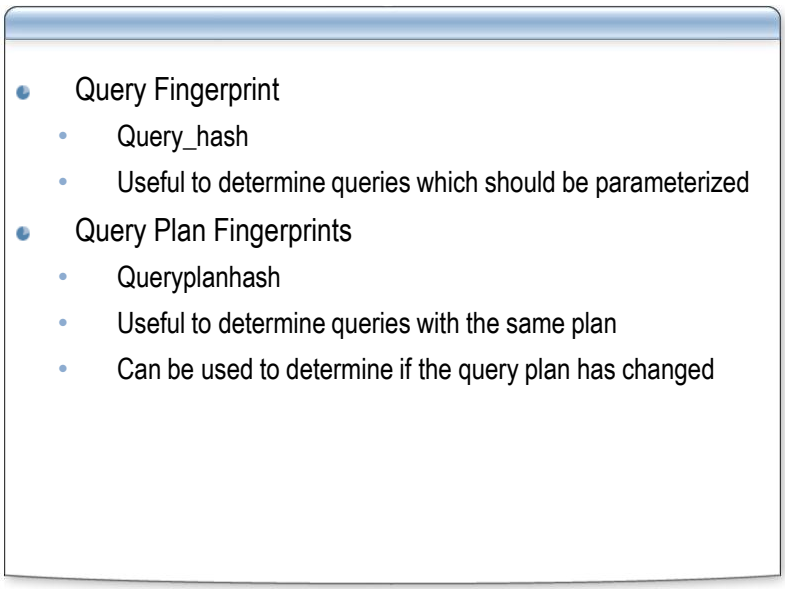
To determine whether your application currently uses parameter markers, you can search the **Text** column in the SQL Server Profiler trace for sp_executesql. However, sp_executesql may be called directly; therefore, all instances will not indicate the use of parameter markers.

### Using Profiler to identify opportunities for sp_executesql

You can use Profiler to find occurrences of dynamic SQL that is being executed using the EXECUTE statement. After you determine where the EXECUTE statement is being used, you should consider replacing it with a call to sp_executesql.

> **Note:** For more information about tuning performance by using sp_executesql, refer to the article *How to Troubleshoot the Performance of Ad-Hoc Queries in SQL Server* at http://support.microsoft.com/kb/243588/EN-US/.

# Query and Query Plan Fingerprints




Query and plan fingerprints are new in SQL Server 2008.

## Query Fingerprint

The query fingerprint is generated by taking the SQL statement and running it through a hash function. If the SQL statements hash to the same

value, they are probably the same SQL statement that only differ by literals. The query fingerprint is exposed as the query_hash column in

sys.dm_exec_requests, and sys.dm_exec_query_stats. The query fingerprint is also exposed as the queryhash attribute of the XML query plan.

The query hash can be used to determine if the plan cache contains similar queries which cannot be auto-parameterized. This situation can occur

more frequently when an application issues ad-hoc SQL statements. For example consider the following SQL statements

Dbcc freeproccache

go

select * from sys.objects where object_id = 100

go

```
select * from sys.objects where object_id = 101
go
select * from sys.objects where object_id = 102
go
```

This is basically the same query with a different literal. If you issue this query, SQL Server will not auto-parameterize the queries. If you look at

the plan cache, you will see this query in cache 3 times. To view the cache run

```
Select * from sys.dm_exec_cached_plans
Cross apply sys.dm_exec_sql_text(plan_handle)
```

Each time the query is issued with a different value for object_id, CPU will need to be consumed to generate a query plan. Additionally the plan

cache can grow and contain lots of single use query plans (the plan can only be reused if the same value for object_id is passed again). The plan

cache consumes pages in the buffer pool and if we have lots of single use plans that reduces the space in the buffer pool to cache query results.

In order to determine if you have queries in cache that should be parameterized run the following query:

```
select query_hash,
        COUNT(*) as num_of_entries,
        SUM(total_worker_time) as total_worker_time,
        MIN(sql_handle) as sample_sql_handle
from sys.dm_exec_query_stats
where query_hash <> 0x0000000000000000
group by query_hash
having COUNT(*) > 1
order by COUNT(*) desc
```

Then take the sql_handle returned from above and run the query below to determine if the SQL statement:

select * from sys.dm_exec_sql_text(<any sample_sql_handle from above>)

go

To parameterize the query you can consider setting the database parameterization option to FORCED. You can also parameterize in query in the

application. Using the example above to parameterize the query run

exec sp_executesql N'select * from sys.objects where object_id = @P1', N'@P1 int', 101

## Query Plan Fingerprints

The query plan fingerprint is generated by taking the query plan for a SQL statement and running it through a hash function. The query plan

fingerprint is exposed as the query_plan_hash column in sys.dm_exec_requests and sys.dm_exec_query_stats. The query plan fingerprint is

also exposed as the queryhash attribute of the XML query plan.

You can use the query plan fingerprint to quickly determine if a query plan has changed. So to determine if the query plan has changed you need

to capture the queryplanhash and then later on you can compare the queryplanhash value again to determine if it has changed.

Demo:

dbcc freeproccache

go

select lastname, firstname, emailaddress

from Person.Contact

where LastName in ('smith','wesson')

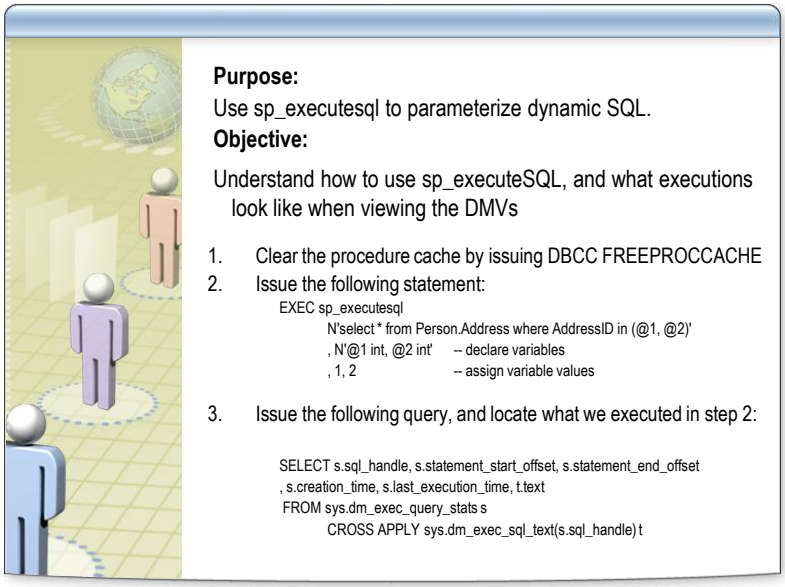
Go

```
-- now get the query plan fingerprint (query_plan_hash) and the query fingerprint
(query_hash)
select query_plan_hash, query_hash,
(total_logical_writes + total_logical_reads + total_physical_reads) as  [total io]
from sys.dm_exec_query_stats
cross apply sys.dm_exec_sql_text(sql_handle)
where text like 'select lastname, firstname, emailaddress
from Person.Contact
where LastName%'



-- add an index


create index mynewindex on person.contact (lastname, firstname, emailaddress)


Run the query again


select lastname, firstname, emailaddress
from Person.Contact
where LastName in ('smith','wesson')
Go
-- now get the query plan fingerprint (query_plan_hash) and the query fingerprint
(query_hash)
select query_plan_hash, query_hash,
(total_logical_writes + total_logical_reads + total_physical_reads) as  [total io]
from sys.dm_exec_query_stats
cross apply sys.dm_exec_sql_text(sql_handle)
where text like 'select lastname, firstname, emailaddress
from Person.Contact
where LastName%'
```

Note the query fingerprint (query_hash) did not change but the query_plan_hash did.

So if you compare query plan hash values before/after applying a service pack, you can isolate which queries had plan changes.

# Demonstration 6: sp_executesql



**Purpose:**

Use sp_executesql to parameterize dynamic SQL.

**Objective:**

Understand how to use sp_executeSQL, and what executions look like when viewing the DMVs

1. Clear the procedure cache by issuing DBCC FREEPROCCACHE
2. Issue the following statement:

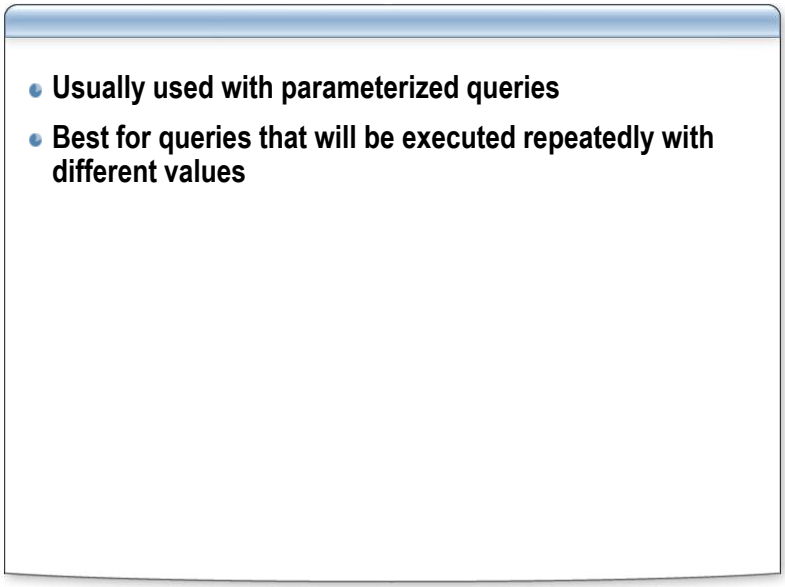
```
EXEC sp_executesql
        N'select * from Person.Address where AddressID in (@1, @2)'
        , N'@1 int, @2 int'     -- declare variables
        , 1, 2                  -- assign variable values
```

3. Issue the following query, and locate what we executed in step 2:

```
SELECT s.sql_handle, s.statement_start_offset, s.statement_end_offset
, s.creation_time, s.last_execution_time, t.text
 FROM sys.dm_exec_query_stats s
        CROSS APPLY sys.dm_exec_sql_text(s.sql_handle) t
```

23

# Prepare and Execute

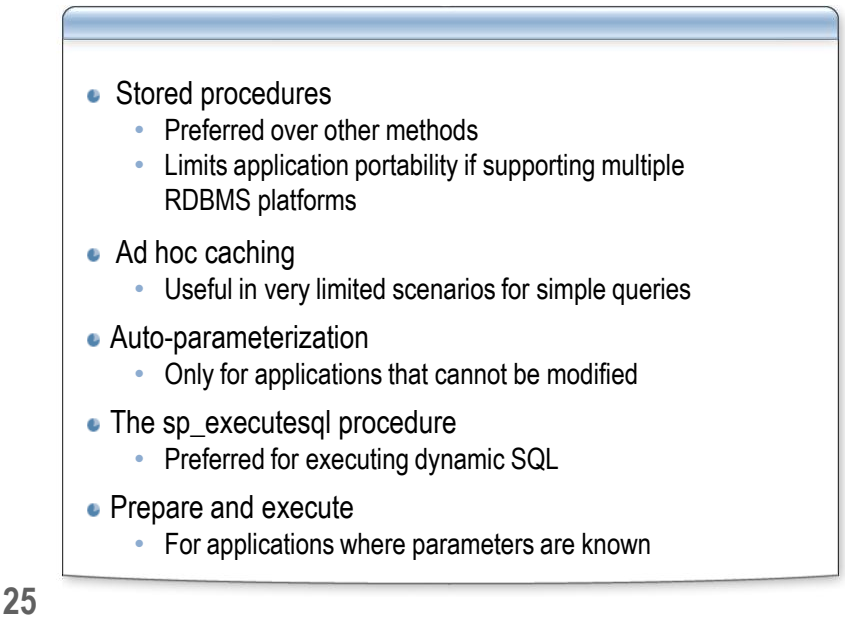



Prepared execution helps reduce the parsing and compiling overhead associated with repeatedly running a T-SQL statement. Prepared execution is commonly used by applications to repeatedly run the same, parameterized SQL statement. The application uses prepared execution to build a character string containing an SQL statement and then runs it in the following two stages:

1. First, it calls SQLPrepare to have the statement parsed and compiled into an execution plan by the Database Engine. At this stage, the SQL Native Client ODBC driver creates a temporary stored procedure from prepared SQL statements. Stored procedures are an efficient way to run a statement multiple times, and are ideal for the execution of a single or a set of statements with different parameter values.
2. Next, it calls SQLExecute for each execution of the prepared execution plan. This saves the parsing and compiling overhead on each execution.

**Caution**: Because of the additional overhead associated with preparing a statement, you should use prepared statements only for those statements that are going to be executed more than three or four times.

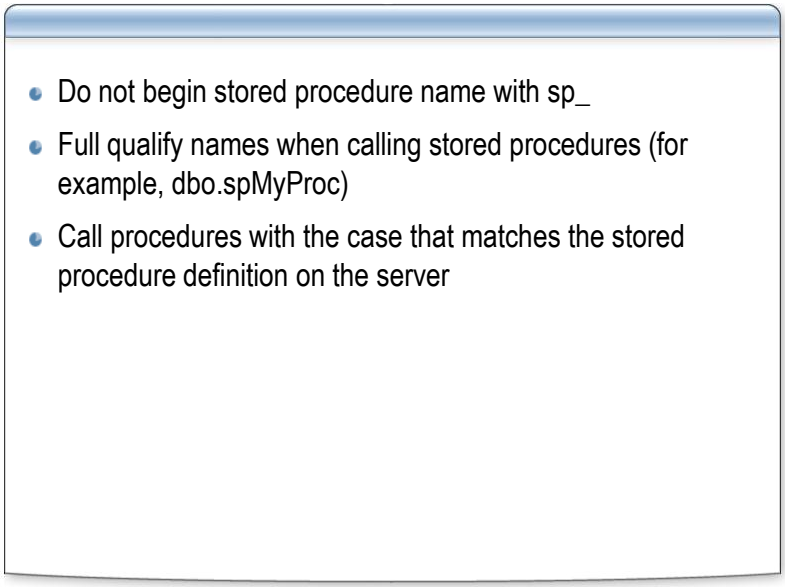
# Selecting a Caching Mechanism




The following table lists the various guidelines that you must consider when deciding whether to use stored procedures or another caching mechanism:

| Method | Guidelines |
|---|---|
| Stored procedures | • Use stored procedures when multiple applications are running batches in which the parameters are known.<br>• Using stored procedures will limit application portability if they are supporting multiple RDBMS platforms.<br>**Note**: Stored procedures are preferred over all other methods. |
| Ad hoc caching | • Do not program to take advantage of ad hoc caching. Ad hoc caching exists to allow SQL to more efficiently deal with existing code that cannot be modified to use a more efficient caching mechanism. |
| Auto-parameterization | • Use auto-parameterization only for applications that cannot be easily modified.<br>• Avoid designing applications that rely on auto-parameterization.<br>**Note**: Auto-parameterization is useful in a very limited number of scenarios. |
| The sp_executesql procedure | • Use sp_executesql when a single user might use the same batch multiple times and the parameters are known. |
| Prepare and execute | • Use prepare and execute when multiple users are running batches in which the parameters are known, or when a single user will use the same batch multiple times. |

# Optimizing Cache Lookup




You can avoid COMPILE blocking by optimizing the ability of SQL Server to efficiently locate stored procedures and lookup the matching plan in cache. To do so, follow the principles listed below.

## Do not begin stored procedure names with sp_

If your stored procedure name begins with the sp_ prefix, and it is not in the master database, **SP:CacheMiss** appears before the cache hit for each execution even if the stored procedure call is owner qualified. The reason for this is that sp_ prefix tells SQL Server that the stored procedure is a system stored procedure. System stored procedures have different name resolution rules. With system stored procedures, SQL server will first look in the master database if the call is not database qualified, and then look in the current database. To avoid this additional work, do not use stored procedure names that begin with sp_.

## Use fully qualified owner names when calling stored procedures

If the object dbo.mystoredproc exists within the schema dbo, and another user whose default schema is *Harry* runs this stored procedure with the command **exec myStoredproc**, the initial cache lookup by object name will fail because the object is not schema qualified. SQL Server does not yet know whether a stored procedure named **Harry.mystoredproc** exists, so SQL Server cannot determine if the cached plan for **dbo.mystoredproc** is the right one to execute. SQL Server then acquires an exclusive compile lock on the procedure and makes preparations to compile the procedure, including resolving the object name to an object ID. Before it compiles the plan, SQL

Server uses this object ID to perform a more precise search of the procedure cache, and it is able to locate a previously compiled plan, even without schema qualification.

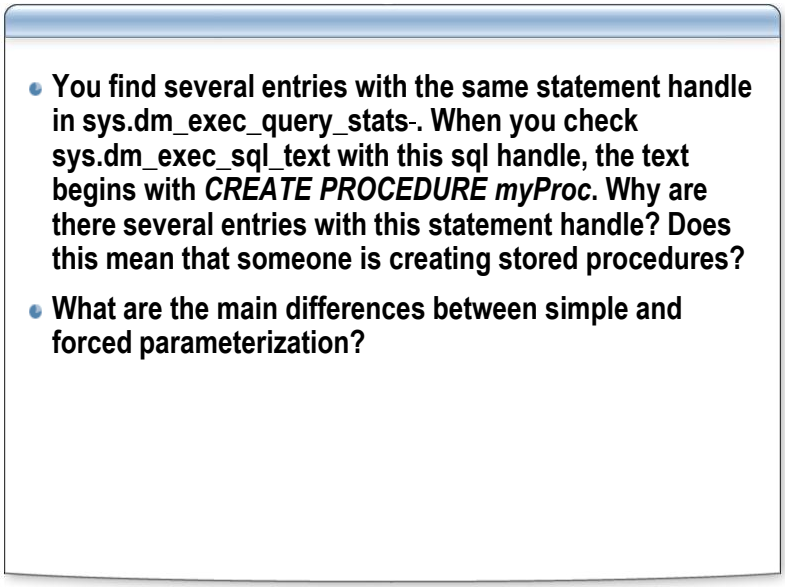
If an existing plan is found, SQL Server reuses the cached plan and does not actually compile the stored procedure. However, the lack of schema qualification forces SQL Server to perform a second cache lookup and to acquire an exclusive compile lock before determining whether the existing cached execution plan can be reused. Acquiring the lock, performing lookups, and other work that is needed to get to this point can introduce a delay that is sufficient for the compile locks to lead to blocking. This is especially true if a large number of users whose default schema is not the schema which contains the stored procedure, simultaneously run the stored procedure without supplying the schema name.

> **Important**: Note that even if you do not see SPIDs waiting for compile locks, lack of schema qualification can introduce both delays in stored procedure execution and unnecessarily high CPU utilization.

## Call procedures with the casing that matches the stored procedure definition on the server

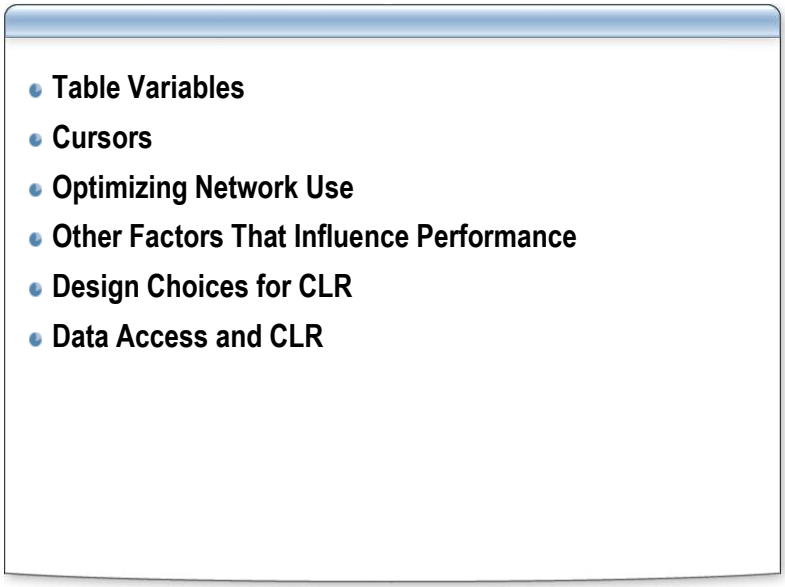
If an owner-qualified procedure is executed with a casing that is different than the one with which it was created, the owner-qualified procedure can get a **CacheMiss** or request a COMPILE lock, but it will eventually use the cached plan. Therefore, the procedure will not actually be recompiled and should not cause much of an overhead. However, the request for a COMPILE lock can get into a **blocking chain** situation if there are many session IDs trying to run the same procedure with a casing that is different than the one with which the procedure was created. This is true regardless of the sort order or collation being used on the server or the database. The reason for this behavior is that the algorithm being used to find the procedure in cache is based on hash values (for performance reasons), which can change if the casing is different. To avoid this situation, you should ensure that the casing of the characters in the stored procedure name matches with the casing of the characters used in the procedure call match.

# Section 2 Review




- You find several entries with the same statement handle in sys.dm_exec_query_stats. When you check sys.dm_exec_sql_text with this sql handle, the text begins with ***CREATE PROCEDURE myProc***. Why are there several entries with this statement handle? Does this mean that someone is creating stored procedures?
- What are the main differences between simple and forced parameterization?

# Section 3: Performance Considerations




## Introduction

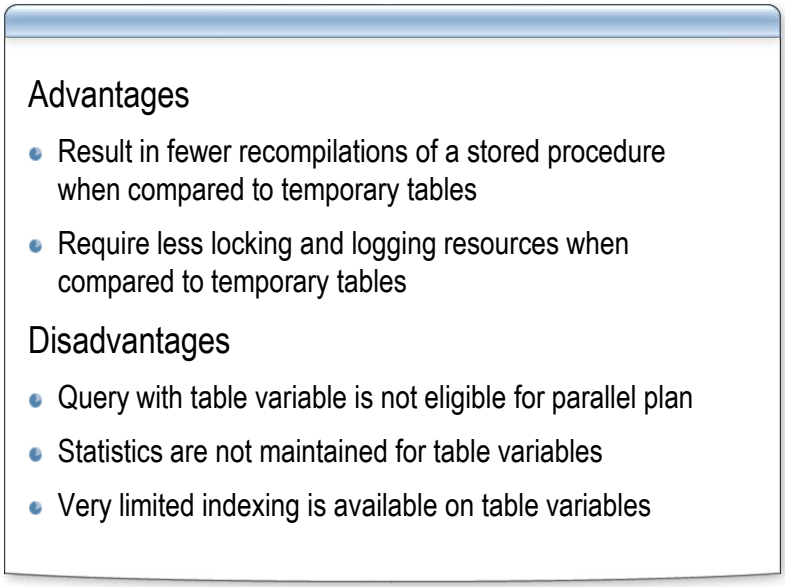
Database and application design are key in determining the performance of SQL Server. Good design makes good performance possible. Likewise, configuration, hardware, or caching mechanisms cannot compensate for poor design. In this section, you will look at key design decisions based on SQL Server features.

## Objectives

After completing this section, you will be able to:

- Explain the advantages and disadvantages of using table variables instead of temporary tables.
- Explain why you should not normally use cursors to return a resultset.
- Optimize network use by minimizing network roundtrips, locks, and client lag time.
- Design databases for performance by applying best practices for normalization, primary and foreign key definition, and indexing.
- Improve performance by appropriately integrating the use of managed code into database applications.
- Determine when to use managed code or T-SQL for data access.

# Table Variables




The general rule of thumb is to use table variables for reasonably small queries and datasets, and to use temporary tables for larger datasets. In deciding whether you should use table variables or temporary tables, you should test and measure both in your environment.

## Advantages of table variables

Table variables have the following advantages:

- When you use table variables, there are fewer stored procedure recompilations.
- Transactions that involve table variables last only for the duration of an update on the table variable. Therefore, table variables require fewer locking and logging resources. Because table variables have limited scope and are not part of the persistent database, they are not affected by transaction rollbacks.

## Disadvantages of table variables

Table variables have the following disadvantages:

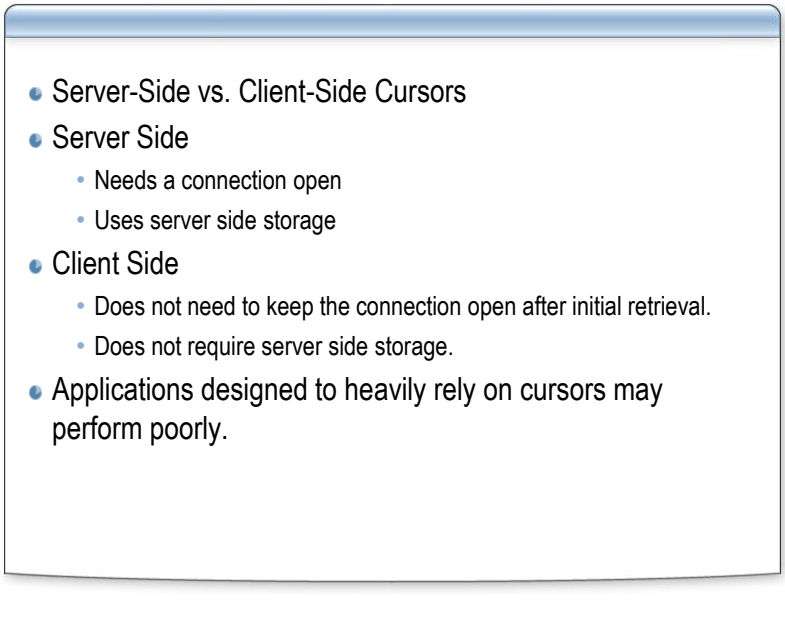
- Queries using table variables are not eligible for parallel plans.
- SQL Server maintains statistics on temporary tables, but not on table variables. Without statistics, SQL Server may choose a poor execution plan for a query.
- You cannot create indexes on a table variable after creation. However, indexes are created at the time of table variable creation if a primary key or unique constraint is created. Consider the following examples:

```
DECLARE @tab TABLE
```

```
(
        col1 INT NOT NULL PRIMARY KEY
,       col2 INT NOT NULL UNIQUE
)

DECLARE @tab1 TABLE
(
        col1 INT NOT NULL
,       col2 INT NOT NULL
,       col3 VARCHAR(20)
        PRIMARY KEY (col1, col2)
,       UNIQUE (col2)
)
```

In both examples above, indexes are created implicitly by the creation of primary keys or unique constraints. But, after the table variable is created, no indexes or constraints can be added to it. This severely limits the number and types of indexes that can be applied to a table variable. As a result of this limitation, performance of queries dealing with large datasets might suffer from the lack of indexes.

# Cursors




Every cursor uses temporary resources to hold its data. These resources can be memory, disk paging files, temporary disk files, or even temporary storage in the database. The cursor is called a *server-side cursor* when these resources are located on the server machine. The cursor is called a *client-side cursor* when these resources are located on the client machine.

## Server-side cursors

When the cursor is a server-side cursor, the server manages the result set by using resources provided by the server machine. The server-side cursor returns only the requested data over the network. A server-side cursor can sometimes provide better performance than the client-side cursor, especially in situations where excessive network traffic is a problem.

Server-side cursors also permit more than one operation on the connection. That is, once the cursor is created, the same connection can be used to make changes to the rows without the need to establish an additional connection to handle the underlying update queries.

However, you must remember that a server-side cursor is, at least temporarily, consuming precious server resources for every active client. You must plan accordingly to ensure that your server hardware is capable of managing all of the server-side cursors requested by active clients. Also, a server-side cursor can be slow because it provides only single row access; a batch cursor is not available.

Server-side cursors may be useful for inserting, updating, or deleting records positionally. Server-side cursors allow multiple active statements on the same connection. SQL Server allows pending results in multiple statement handles.

### Advantages to using server-side cursors

- **Memory usage**. The client does not need to cache large amounts of data or maintain information about the cursor position because the server is doing that.
- **Performance**. When only some of the data in the result set is accessed, or the data is accessed a few times only, a server-side cursor provides enhanced performance because it minimizes the network traffic.
- **Additional cursor types**. Both keyset and dynamic cursors are available for server-side cursors.
- **Positioned updates**. Server-side cursors support direct positioned updates, whereas ODBC simulates positioned cursor updates by generating a SQL search and update statement. Direct positioned updates are not only faster, but they also avoid the risk of unintended update collisions.

## Client-side cursors

When the cursor is a non-keyset client-side cursor, the server sends the entire result set across the network to the client machine. The client machine provides and manages the temporary resources needed by the cursor and the result set. The client-side application can browse through the entire result set to determine the rows that it requires.

Static client-side cursors might place a significant load on your workstation if they include too many rows. While all of the cursor libraries are capable of building cursors with thousands of rows, applications designed to fetch such large rowsets might perform poorly. However, there are exceptions. For some applications, a large client-side cursor may be perfectly appropriate and performance may not be an issue.

One obvious benefit of the client-side cursor is quick response. After the result set has been downloaded to the client machine, browsing through the rows is very fast. An application is generally more scalable with a client-side cursor, because the resource requirements of the cursor are placed on each separate client and not on the server.

### Cursors with ADO.NET

Most of the features of the ADO Recordset are split into three key objects in ADO.NET:

- **DataReader**: The ADO.NET DataReader object is designed to be a server-side, forward-only, read-only cursor.
- **DataSet**: The ADO.NET DataSet object is a disconnected storage tool for rowsets. It stores records without holding a connection to a data source. The DataSet is similar to the ADO Recordset object with its CursorType set to adOpenStatic and CursorLocation set to adUseClient, when it has been disconnected from its associated Connection object (via the ActiveConnection property of Recordset).

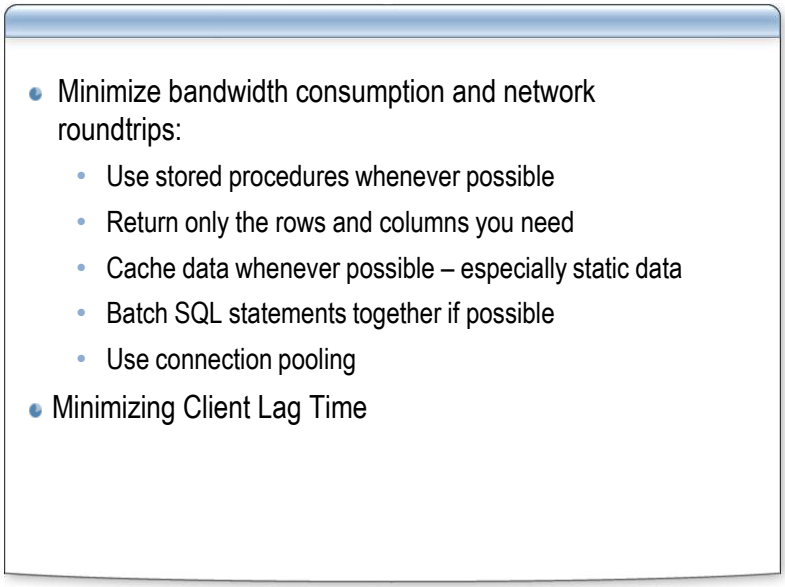
- **DataAdapter**: The ADO.NET DataAdapter object is a bridge between the connection and DataSet objects. It can load a DataSet from a data source by using a connection, and it can update a data source with the changes stored in a DataSet. When the underlying data has been changed between the time the DataSet was filled and the DataAdapter attempts to merge the updates back into the database, a DBConcurrencyException is thrown. This exception can be caught and handled by the application.

The current versions of ADO.NET do not expose a multidirectional, scrollable, updateable, server-side cursor. As a result, positioned updates are not available in ADO.NET.

## Cursor performance

Cursors require resources to be set aside and maintained for dealing with the results, and may cause SQL to constantly negotiate additional locking. Cursors may also require additional roundtrips to the server. To optimize performance, you should minimize the use of cursors and use set-based operations, whenever possible.

# Optimizing Network Use




When optimizing the performance of a SQL-based application, you must consider every resource. It is unlikely for the consumers of an enterprise application to be logged onto the SQL Server directly; therefore, the network can become a performance bottleneck. Keep the following guidelines in mind when looking for ways to optimize network use.

## Minimize network roundtrips

If an application uses a loop, you should consider putting the loop inside a single batch. Often, an application contains a loop that contains a parameterized query that is executed many times and requires a network roundtrip between the computer running the application and SQL Server each time it is run. To avoid this, you can create a single, more complex query by using a temporary table. Then, only one network roundtrip would be necessary, and the query optimizer can better optimize the single query.

## Use stored procedures whenever possible

Stored procedures offer a way to reduce network traffic in some situations. Consider the following:

- Queries can be quite long and complex. By comparison, stored procedure names are relatively short.
- Applications often take different execution paths based on the results of a query. The results of a query may lead to other queries being issued. Each of these queries requires a network roundtrip and transmits additional results to the client. If the logic is moved to the stored procedure, then all of this can be performed with one network roundtrip.

- When logic is performed in an application, each intermediate resultset must be sent to the client application. These resultsets can be quite large and can consume large amounts of network resources. If only the end result is necessary, then a stored procedure can deal with these intermediate results on the server without transmitting large resultsets to the client. When all of the data is processed and shaped, a single resultset can be sent to the client application.

### Cache data whenever possible

Applications often use static data. In n-tiered architectures such as Web applications, several users may need to access portions of the same static data. Caching the data on the Web servers allows this to happen efficiently and eliminates repeated network roundtrips to the SQL Server to retrieve the same data.

### Batch SQL statements together whenever possible

Each batch represents an additional network roundtrip. If stored procedures cannot be used, often you can batch several statements to minimize the number of network roundtrips.

### Use connection pooling

Each login to the SQL Server requires additional resources both on the SQL Server itself and on the network. Connection pooling allows connections to be reused, instead of disposed of after each user finishes the required tasks. This not only allows minimizing the number of connections necessary, but also allows minimizing the network resources required to repeatedly establish connections and log on.

### Minimizing client lag time

Client lag time is the time period during which the client processes the results that it receives. If you consider an application that queries for an initial dataset, and then uses this data for several subsequent queries, you can quickly see how this can affect the process. If the client needs an extra ten milliseconds (ms) to handle a resultset, it can add a significant amount of time to the transaction latency time.

For each dataset, SQL Server fills the first tabular data stream (TDS) packet, sends it to the client, and then waits for the client to process the results. During the time in which the client is processing the results (client latency time), SQL Server continues to hold a shared page lock on that page on which it was processing. This shared lock can block a user who is attempting to complete an order. Large numbers of such delays can add up to a poorly performing system.

### How to verify network or client-side latency

SQL Server Profiler has a **SQLTransaction** event. The **EventSubClass** column of this event indicates whether it was a **Begin**, **Commit**, or **Rollback** event. SQL Server Profiler also shows you the statements that were sent within that transaction. You should check for **Batch:Starting/Completed** or **RPC Starting/Completed** events, or BEGIN, COMMIT, or ROLLBACK in the EventSubClass column, and verify that transactions are

held open only for the minimum required time. Remember that at some point, you might need to separate batches to enable higher levels of concurrency. A good experiment is to delete 20 percent of the rows from a million row table, and look at the amount of time that is consumed trying to delete all the rows compared to the amount of time consumed deleting smaller chunks. You might want to try deleting 250 rows at a time for the purpose of comparison.

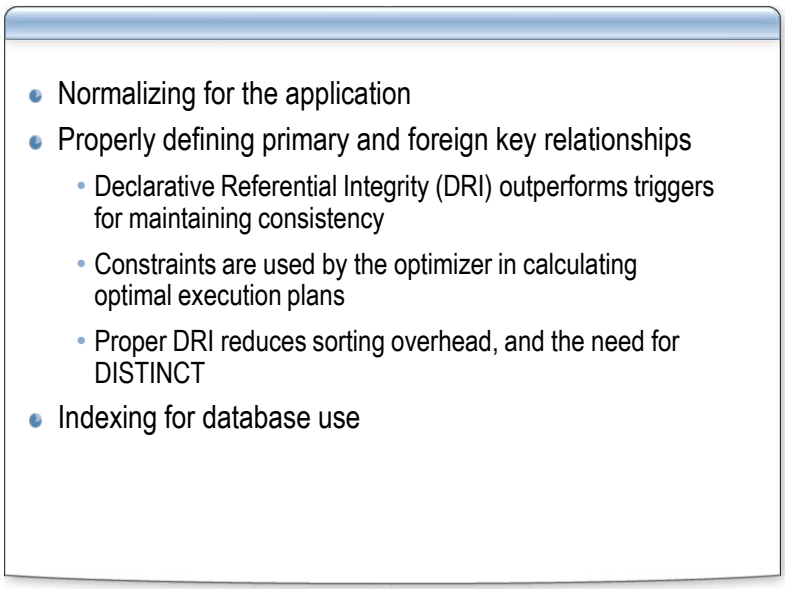
Another way to verify network latency or client-side latency is to run the following script:

```
SET STATISTIC TIME ON
<query>
SET STATISTICS TIME OFF
```

Compare the time returned by the STATISTICS TIME to the **Duration** column in SQL Server Profiler and note the difference. The duration time in SQL Server Profiler shows the total time, including network time and the time that the client spent processing the results.

> **Note**: For more information about optimizing network use, refer to the following articles:
>
> Improving ADO.NET Performance at
> http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/ScaleNetChapt12.asp
>
> How to troubleshoot the performance of Ad-Hoc queries in SQL Server at
> http://support.microsoft.com/kb/243588/EN-US/
>
> Improving SQL Server Performance at
> http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt14.asp

# Other Factors that Influence Performance




Database design affects the performance and scalability of database applications. Consider the following design factors as they apply to your application.

## Normalization and denormalization

You can achieve a logical database design by applying normalization rules to your design. Normalization provides several benefits such as reducing data redundancy and maintaining data consistency. When you reduce redundant data, you can create narrow and compact tables. However, over-normalization of a database schema may affect performance and scalability. Obtaining the right degree of normalization involves tradeoffs. On one hand, you want a normalized database to limit data duplication and maintain data integrity; on the other hand, it might be harder to program against fully normalized databases, which can affect performance.

For example, addresses are one part of a data model that is typically denormalized. Because many systems store multiple addresses for companies or people over long periods of time, it is relationally correct to have a separate address table and to join to that table to get the applicable address. However; it is a common practice to keep the current address duplicated in the person table, or even to keep two addresses because this type of information is fairly static and is accessed often. The performance benefits of avoiding the extra join generally outweigh the consistency problems in this case.

The following denormalization approaches can help:

- Start with a normalized model, and then denormalize, if necessary. Do not start with a denormalized model, and then normalize it. Typically, each denormalization

requires a compensating action to ensure data consistency. The compensating action might adversely affect performance.

- Avoid highly abstracted object models. Such models might be extremely flexible, but they are often overly complex, difficult to understand, and can result in too many self joins. For example, many business objects can be modeled by using an Object table, an Attributes table, and a Relationship table. This object model is very flexible, but the number of self joins, alias joins, and joins becomes so cumbersome that it is not only difficult to write queries and understand them, but performance and scalability also suffer. For an abstract object model, you should try to find some common object types that can be used as subtypes under the generic object type, and then try to find the best balance between flexibility and performance.

## Defining primary and foreign key relationships

Primary keys and foreign key relationships that are correctly defined help ensure that you can write optimal queries. One common result of incorrect relationships is that DISTINCT clauses are added to eliminate redundant data from resultsets.

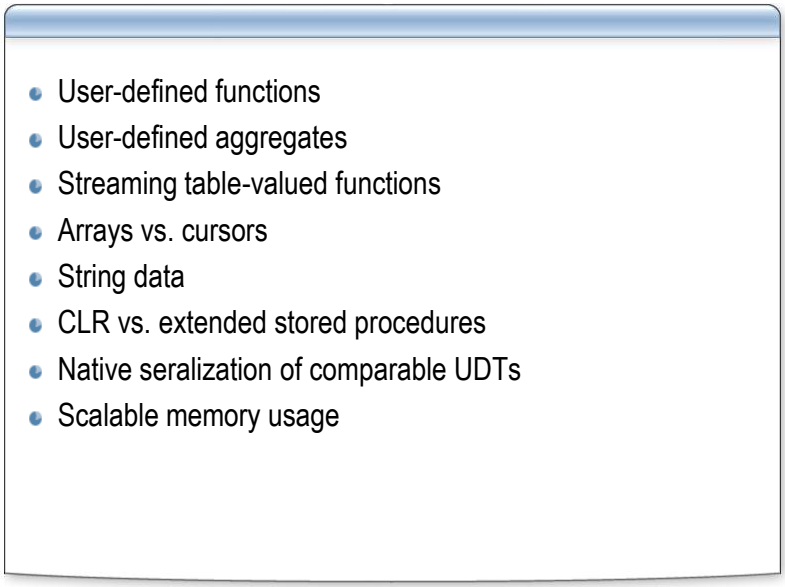
When primary and foreign keys are defined as constraints in a database schema, the server can use that information to create optimal execution plans.

Declarative Referential Integrity (DRI) performs better than triggers, and it is also easier to maintain and troubleshoot than triggers. The server checks for DRI before it performs the actual data modification request. When you use triggers, the data modification requests are inserted into the inserted and deleted temporary system tables, and the trigger code is run. Depending on the trigger code, the final modifications are then either made or not made.

## Indexing for database use

Creating and maintaining an index structure incurs cost. Having a large number of indexes on a table might result in faster SELECT statements but slower INSERT, UPDATE, and DELETE statements. The performance overhead will vary by application and database. If you have a large number of indexes on a table, you increase the chance of the query optimizer choosing a suboptimal index for a query plan. Additionally, if a column is included in several indexes, then the values must be maintained in every index containing that column whenever an INSERT or UPDATE is performed on that column.

# Design Choices for CLR




Beginning with SQL Server 2005, you can create programming objects in any .NET-compliant language. This adds a great deal of power to SQL server-side programming. Remember the following items when considering whether or not to create CLR objects.

## User-defined functions

CLR functions benefit from quicker invocation paths than those of T-SQL user-defined functions. Additionally, managed code has a decisive performance advantage over T-SQL in terms of procedural code, computation, and string manipulation. CLR functions that are computing-intensive and that do not perform data access are better written in managed code. However, T-SQL functions perform data access more efficiently than CLR integration.

## User-defined aggregates

Managed code can significantly outperform cursor-based aggregation. Managed code generally performs slightly slower than built-in SQL Server aggregate functions. For this reason, you should use built-in aggregate functions when available for the desired aggregation. In cases where the needed aggregation is not natively supported, you should consider using a CLR user-defined aggregate over a cursor-based implementation.

## Streaming table-valued functions

Applications often need to return tables as a result of invoking functions. For example, an application might need to read tabular data from a file as part of an import operation, and convert comma-separated values (CSVs) into a relational representation. Typically, you can accomplish this by materializing and populating the result table before it can be

consumed by the caller. The integration of CLR into SQL Server introduces a new extensibility mechanism called a streaming table-valued function (STVF). Managed STVFs perform better than comparable extended stored procedure implementations.

An STVF is a managed function that returns an **IEnumerable** interface. IEnumerable contains methods to navigate the resultset returned by the STVF. When the STVF is invoked, the returned IEnumerable is directly connected to the query plan. The query plan calls IEnumerable methods when it needs to fetch rows. This iteration model enables results to be consumed immediately after the first row is produced, instead of waiting until the entire table is populated. It also significantly reduces the memory consumed by invoking the function.

### Arrays vs cursors

When T-SQL cursors must traverse data that is more easily expressed as an array, you can use managed code with significant performance gains.

### String data

SQL Server character data, such as **varchar**, can be of the type **SqlString** or **SqlChars** in managed functions. **SqlString** variables create an instance of the entire value in memory. **SqlChars** variables provide a streaming interface that you can use to achieve better performance and scalability by not requiring an instance of the entire value in memory. This becomes particularly important for large object (LOB) data. Additionally, you can access server XML data through a streaming interface returned by **SqlXml.CreateReader**.

### CLR vs extended stored procedures

The **Microsoft.SqlServer.Server** application programming interfaces (APIs) that enable managed procedures to send resultsets back to the client perform better than the Open Data Services (ODS) APIs used by extended stored procedures. In addition, the **System.Data.SqlServer** APIs support data types such as **xml**, **varchar(max)**, **nvarchar(max)**, and **varbinary(max)** introduced in SQL Server 2005, but the ODS APIs have not been extended to support the new data types.

With managed code, SQL Server manages the use of resources such as memory, threads, and synchronization. This is because the managed APIs that expose these resources are implemented on top of the SQL Server Resource Manager. Conversely, SQL Server does not have any view or control over the resource usage of the extended stored procedure. For example, if an extended stored procedure consumes too much CPU or memory resources, there is no way to detect or control this with SQL Server. With managed code, however, SQL Server can detect that a given thread has not yielded for a long period of time, and then force the task to yield so that other work can be scheduled. Consequently, using managed code provides for better scalability and system resource usage.

Managed code may incur additional overhead necessary to maintain the execution environment and perform security checks. For example, this is the case when the managed code is running inside SQL Server, and numerous transitions from managed to

native code are required (because SQL Server needs to do additional maintenance on thread-specific settings when it is moving out to native code and back). Consequently, extended stored procedures can significantly outperform managed code that is running inside SQL Server for cases where there are frequent transitions between managed and native code.

> **Note**: We recommend that you do not develop new extended stored procedures, because this feature has been deprecated.

### Native serialization for user-defined types

User-defined types (UDTs) are designed as extensibility mechanisms for the scalar type system. SQL Server implements a serialization format for UDTs called ***Format.Native***. During compilation, the structure of the type is examined to generate Microsoft Intermediate Language (MSIL) code that is customized for that particular type definition.

Native serialization is the default implementation for SQL Server. User-defined serialization invokes a method that is defined by the type author to perform the serialization. For best performance, you should use Format.Native serialization, whenever possible.

### Normalization of comparable UDTs

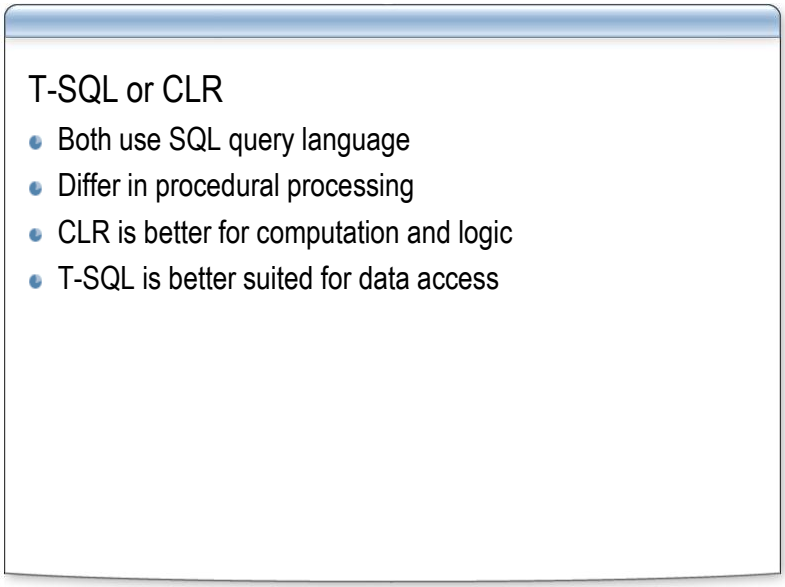
Relational operations, such as sorting and comparing UDTs, are performed by comparing the binary representation of the value. This is accomplished by storing a normalized (binary ordered) representation of the state of the UDT on disk.

Normalization has two benefits. It makes the comparison operation considerably less expensive by avoiding the construction of the type instance and the method invocation overhead. In addition, it creates a binary domain for the UDT, enabling the construction of indexes and histograms for values of the type. Consequently, normalized UDTs have very similar performance profiles to the native built-in types for operations that do not involve method invocation.

### Scalable memory usage

In order for managed garbage collection to perform and scale well in SQL Server, you should avoid large, single allocations. Allocations that are greater than 88 kilobytes (KB) will be placed on the Large Object Heap, which will adversely affect the garbage collection performance and scalability. So, if you need to allocate a large multi-dimensional array, it is better to allocate a jagged (scattered) array.

## Data Access and CLR



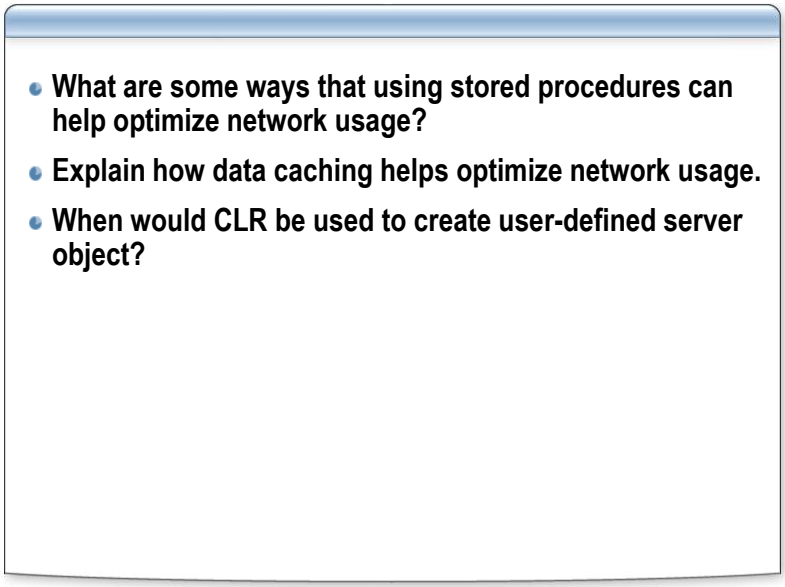

CLR languages are procedural languages that can use APIs to retrieve and manipulate data from a SQL Server database. T-SQL is a procedural extension to the SQL query language and is the native language of SQL Server. Therefore, you can use both CLR and T-SQL to access data, and use this data in procedural code, but each has an advantage in a distinct type of processing.

In T-SQL, query language statements, such as SELECT, INSERT, UPDATE, and DELETE are simply embedded within procedural code. On the other hand, managed code uses the ADO.NET data access provider for SQL Server (SqlClient). In this approach, all query language statements are represented by dynamic strings that are passed as arguments to methods and properties in the ADO.NET API. This difference causes data access code written by using the CLR to be more verbose than code written in T-SQL. More importantly, because the SQL statements are encoded in dynamic strings, they are not compiled or validated until they are executed. This can have an adverse affect on both the debugging of the code and its performance. However, the database programming model with ADO.NET is very similar to that used in the client or middle tiers, which makes it easier for developers familiar with this environment to move the code between the tiers and to leverage existing skills.

Managed code has a decisive performance advantage over T-SQL with respect to most procedural computation, but for data-access, T-SQL generally fares better. Therefore, a good general rule is that computation-intensive and logic-intensive code is a better choice for CLR implementation than data-access intensive code. You should perform CREATE, RETRIEVE, UPDATE, and DELETE operations by using T-SQL procedures.

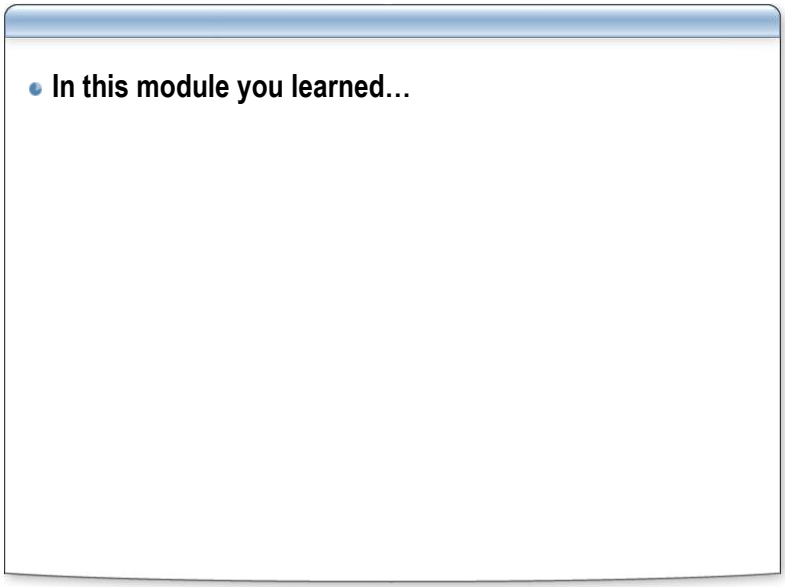
**Note: For more** information, refer to the topic *Using CLR Integration in SQL Server 2005* at http://msdn2.microsoft.com/en-us/library/ms345136.aspx.

# Section 3 Review




- What are some ways that using stored procedures can help optimize network usage?
- Explain how data caching helps optimize network usage.
- When would CLR be used to create user-defined server object?

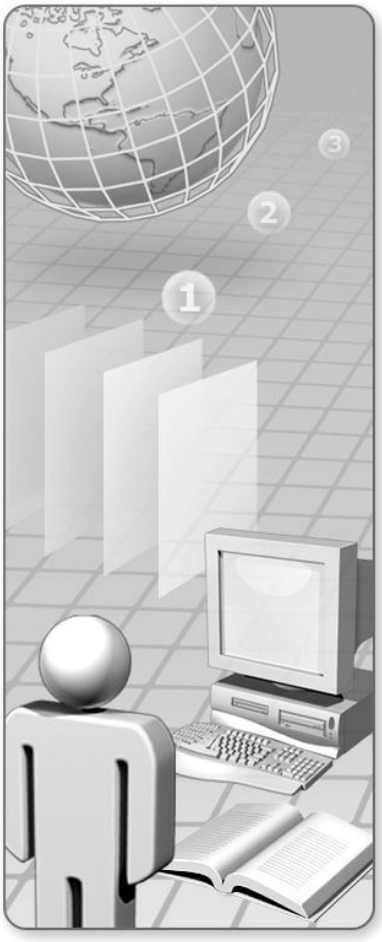
# Module Summary



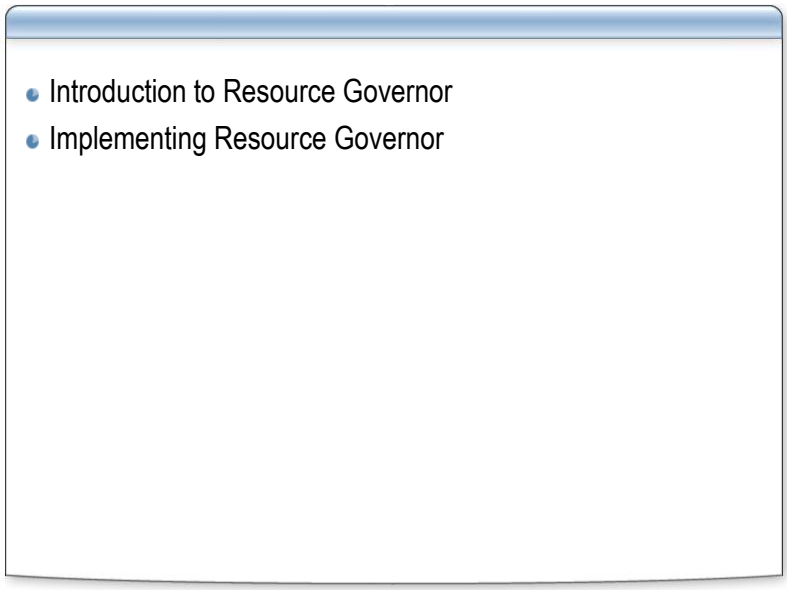

In this module, you learned how to:

- Optimize the storage and compilation of stored procedures.
- Optimize queries and caching.
- Design databases for performance.

# Module 7: Resource Governor



# Module Overview




## Introduction

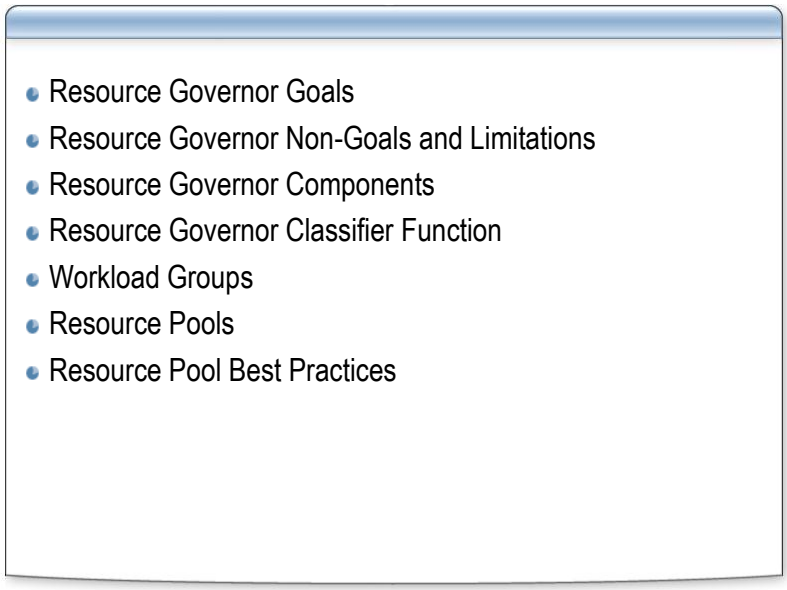
Resource Governor is useful when you have different workloads using the same SQL Server instance. The goal of Resource Governor is to provide predictable execution of workloads. Resource Governor is used to control memory and CPU usage by workloads.

## Objectives

After completing this module, you will be able to:

- Explain how you can use Resource Governor to manage SQL Server workload and resources by specifying limits on resource consumption by incoming requests.
- Implement Resource Governor.
- Troubleshoot SQL Server workload and shared resource allocations for CPU and memory by using Resource Governor.

# Section 1: Introduction to Resource Governor




## Introduction

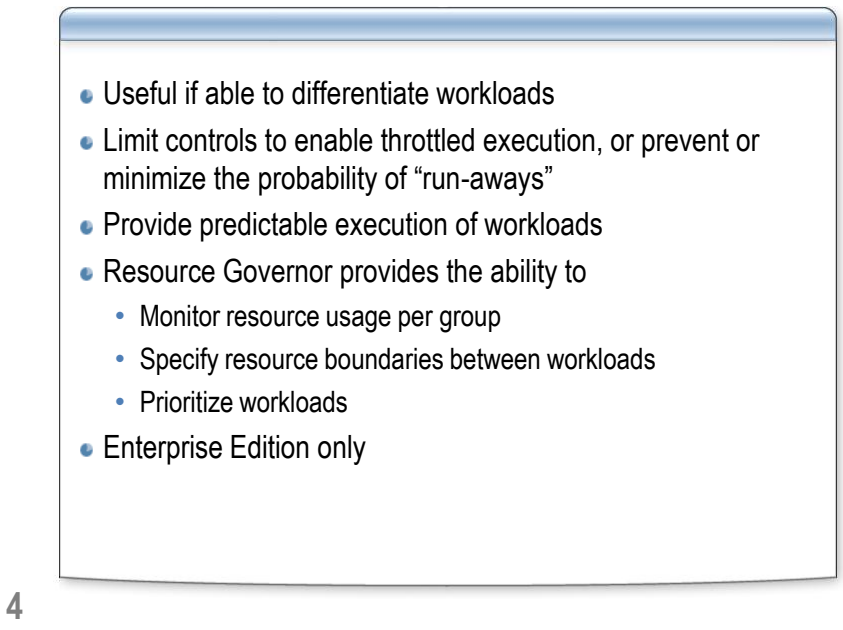
This section describes the goals and limitations of Resource Governor. The section also introduces the components of Resource Governor.

## Objectives

After completing this section, you will be able to:

- Explain the primary purpose of Resource Governor.
- Describe the limitations of Resource Governor.
- Identify the various components of Resource Governor.
- Explain the best practices for using the Resource Governor classifier user-defined function (UDF) for implementing classification rules.
- Create a new workload group.
- Define the minimum resource availability of a resource pool and the maximum size of the resource pool for each resource.
- Employ best practices for creating and managing resource pools.

# Resource Governor Goals



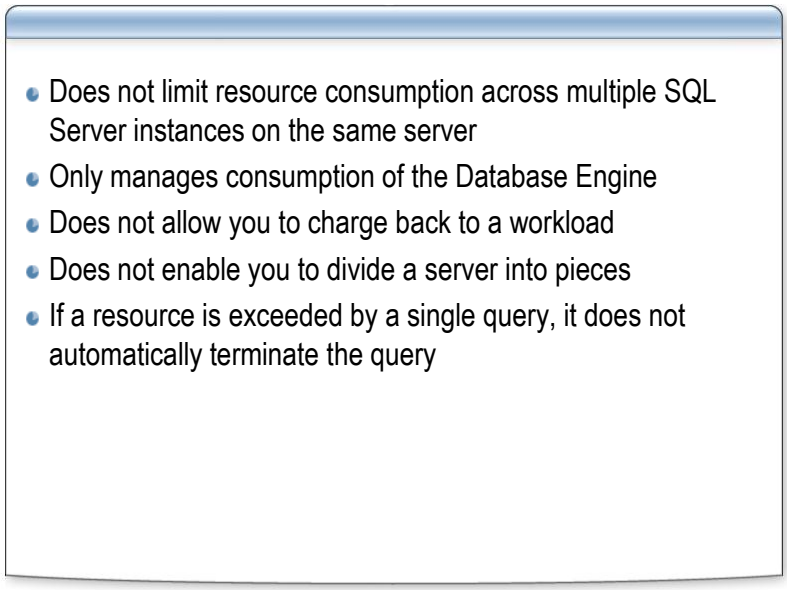

Resource Governor is a new technology in SQL Server 2008 that enables you to manage SQL Server workload and resources by specifying limits on resource consumption by incoming requests.

In an environment where multiple distinct workloads are present on the same server, Resource Governor enables you to differentiate between these workloads and allocate shared resources as they are requested, based on the limits that you specify. For instance if all connections to SQL Server are made with the same login, you will be unable to differentiate workloads and therefore Resource Governor should not be enabled. These resources are CPU and memory.

In order to leverage Resource Governor, you need to be able to differentiate between workloads on the same SQL Server instance. If you cannot do this differentiation, you probably should not implement Resource Governor.

After you have differentiated between the workloads, you can use Resource Governor to prevent one workload from monopolizing resources within a SQL Server instance. You can also prioritize the workload.

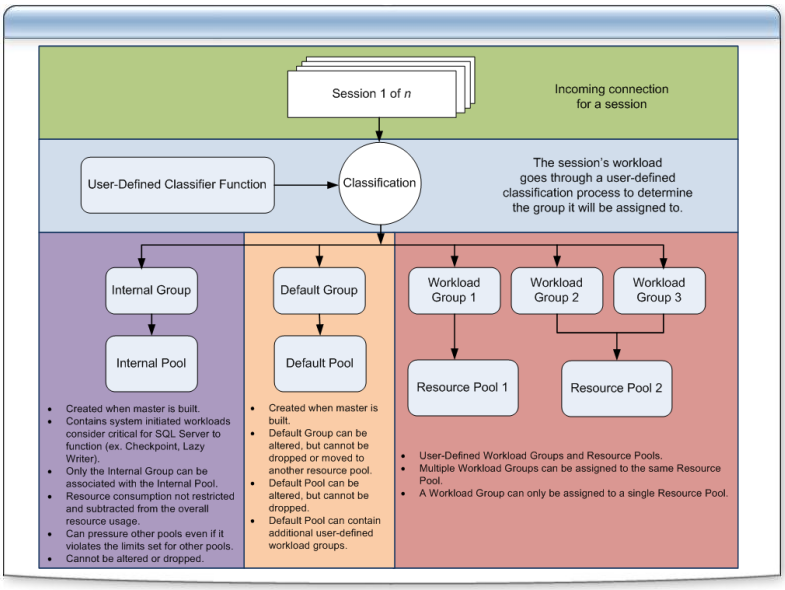
# Resource Governor Limitations




Resource Governor does not provide the following functionalities:

- It does not prevent you from controlling resources across multiple SQL Server instances on the same server.
- It only manages the resource consumption of the Database Engine. You cannot use Resource Governor to manage workload within SQL Server Analysis Services (SSAS), SQL Server Integration Services (SSIS), or SQL Server Reporting Services (SSRS).
- It does not provide any mechanism to charge system resources back to workloads. For example, if each workload was a different department, you cannot use Resource Governor to charge each department for their utilization of resources within the SQL Server instance.
- You cannot use Resource Governor to divide a server into pieces. For example, you cannot use Resource Governor to specify that certain CPUs are dedicated to a specific workload.
- If a query exceeds the limit imposed on the workload, an event is raised but the query is not automatically cancelled and it will continue to run.

# Resource Governor Components



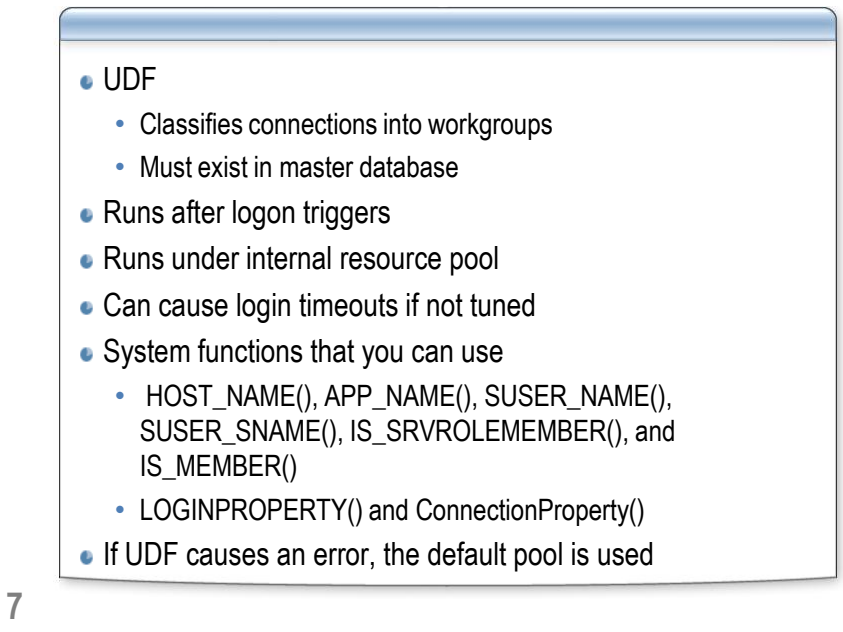

The Resource Governor consists of 3 components:

Classifier Function: used to separate sessions into workloads.

Workload groups: assign workload groups to resource pools.

Resource Pools: used to assign cpu and memory usage to workloads.

# Resource Governor Classifier Function




There are internal rules that classify incoming requests and route them to a workload group. Resource Governor supports a classifier user-defined function (UDF) for implementing classification rules.

The classifier function is a UDF which runs after a connection is established. Therefore, you should keep it short and simple to prevent login timeouts. The classifier function determines which workload a session belongs to.

The classifier UDF must exist in the master database. If the classifier UDF is modified or replaced, existing sessions are not re-classified. To re-classify existing sessions, you must first disconnect and reconnect them. If the classifier function results in an error, the error will be written to the SQL Server error log and the default resource pool will be used.

## Best practices

When using the Resource Governor classifier UDF, you should:

- Avoid complex logic.
- Avoid accessing user databases because if the database is suspect, you cannot login.
- Avoid using table lookups because it might lead to blocking.
- Have a function return default instead of null if the function fails to classify a session into a workgroup. The following is an example:

```
CREATE FUNCTION rgclassifier_v1() RETURNS SYSNAME
WITH SCHEMABINDING
AS
BEGIN
```

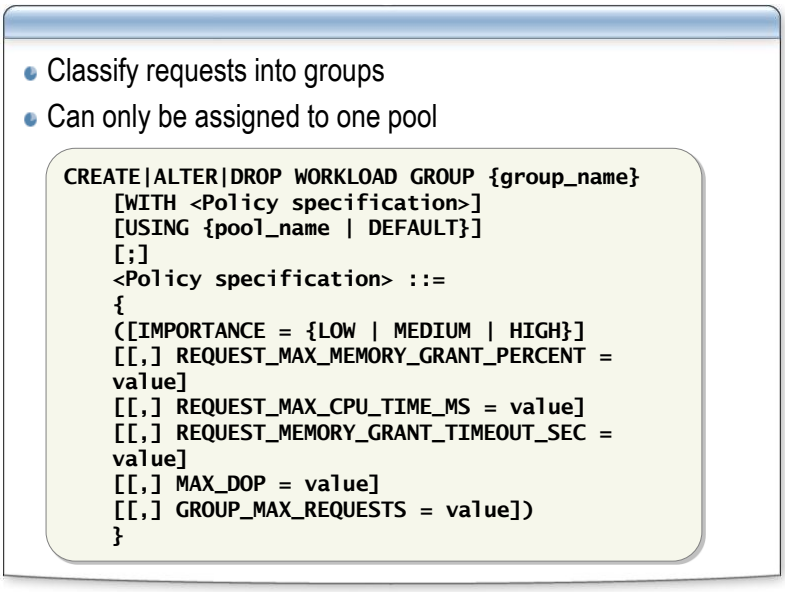
```
   -- Admins
   IF (SUSER_NAME() = 'sa')
          RETURN N'groupAdmin'
   -- Adhoc users
   IF (APP_NAME() LIKE '%MANAGEMENT STUDIO%') OR (APP_NAME() LIKE '%QUERY
ANALYZER%')
          RETURN N'groupAdhoc'
   -- Reporting server application
   IF (APP_NAME() LIKE '%REPORT SERVER%')
          RETURN N'groupReports'
   -- Anything else should be assigned to default
   RETURN N'default'
END
```

**Caution:** When you create a function that returns a property used for classification, you must consider whether or not the property is secure. If not, you must assess the risk of using the attribute. The HOST_NAME() and APP_NAME() functions return properties that are not secure. For example, APP_NAME() can return any value that is provided in an application connection string. Whatever information is not coming from SQL Engine (and is under user control to provide) should be considered unsecure.

System functions that can be used in the classifier UDF are:

```
       HOST_NAME()
       | APP_NAME()
       | LOGIN_NAME()
       | SUSER_NAME()
       | SUSER_SNAME()
       | PROTOCOL_TYPE()
       | PAYLOAD_TYPE()
       | AUTH_SCHEME()
       | SERVER_NET_ADDRESS()
       | SERVER_TCP_PORT()
       | CLIENT_NET_ADDRESS()
       | IS_SRVROLEMEMBER()
       | IS_MEMBER()
       | DEFAULT_DATABASE()
}
```

# Workload Groups



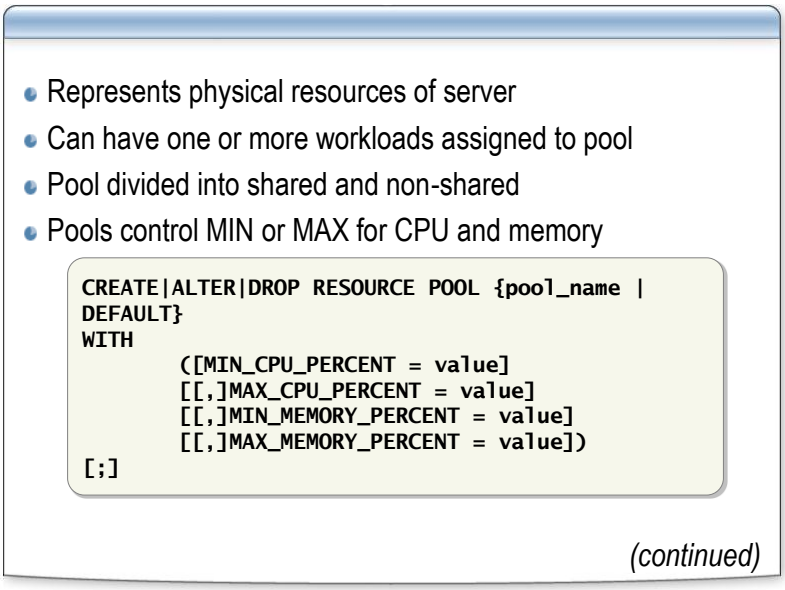

A workload helps you separate requests into groups. If you do not specify a resource pool, the workload uses the default resource pool.

The CREATE WORKLOAD GROUP command creates a Resource Governor workload group and associates it with a Resource Governor resource pool. This command has the following arguments:

| Argument | Description |
|---|---|
| IMPORTANCE (LOW, MEDIUM, HIGH) | Determines how the requests distribute CPU between groups and who gets the first chance to acquire memory. Importance should be treated differently from priority, because when memory is acquired, you do not preempt the query to free up memory. However, you should rebalance CPU consumption by queries in different groups if there are requests with different importance. |
| REQUEST_MAX_MEMORY_GRANT_PERCENT | Specifies the maximum memory size of a single request in the resource pool (expressed as a percentage of the resource pool that the group is in).<br>If a query requests an amount of memory that is larger than specified, it will proceed by using the maximum amount if it is available. If the query cannot run with that amount, the query fails. |
| REQUEST_MAX_CPU_TIME_MS | Specifies the maximum CPU consumption, in seconds, by a single request. When this limit is |

| Argument | Description |
|---|---|
| | exceeded, an event is generated but the query continues to run. |
| REQUEST_MEMORY_GRANT_TIMEOUT_SEC | Specifies the maximum time a query will wait for a memory grant. If the time out is reached the query may not fail. Instead the query will run using the minimum memory grant. |
| MAX_DOP = value | Specifies the maximum degree of parallelism (DOP) for parallel requests. This value must be either 0 or a positive integer value from zero (the default) through 64. Setting this value to zero causes it to use the global setting. MAX_DOP is handled as follows:<br>• MAX_DOP as a query hint is effective as long as it does not exceed the workload group MAX_DOP.<br>• MAX_DOP as a query hint always overrides sp_configure max DOP in SQL Server 2005.<br>• Workload group MAX_DOP overrides sp_configure max DOP.<br>• If the query is marked as serial at compile time, it cannot be changed back to parallel at run time regardless of the workload group or sp_configure setting.<br>• After DOP is configured, it can only be lowered on grant memory pressure. Workload group reconfiguration is not visible while waiting in the grant memory queue. |
| GROUP_MAX_REQUESTS | Specifies the maximum number of requests in the specified group. This is a simple limit on simultaneously active queries in the group. |

# Resource Pools




A resource pool represents the physical resources of the server. A resource pool has two parts—one that does not overlap with other resource pools and the other one that is shared with other resource pools. The part that does not overlap with other resource pools provides a minimum reservation of resources. The part that is shared with other resource pools provides maximum possible resource consumption. In the current model, the resource pool is specified by four numbers:

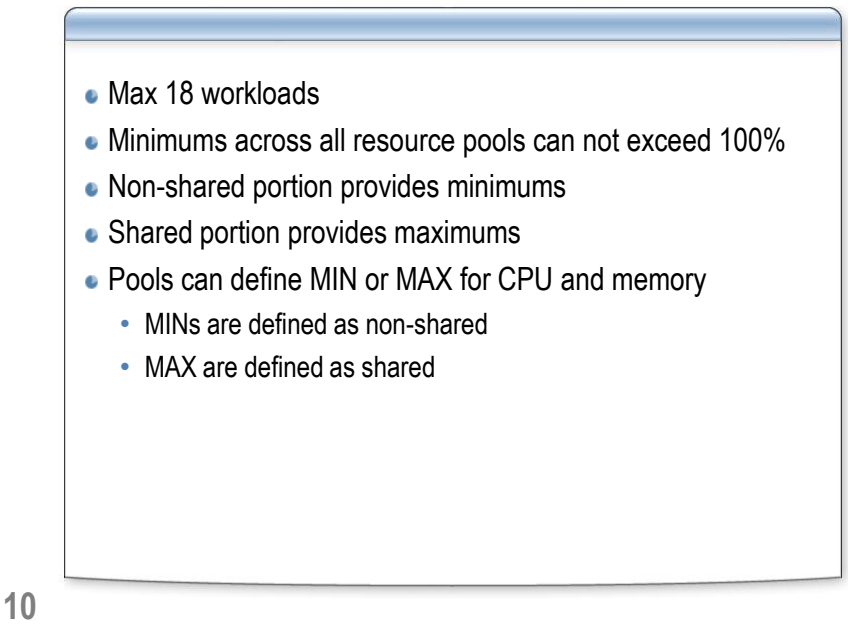
- MIN or MAX for CPU
- MIN or MAX for memory

These numbers define the minimum guaranteed resource availability of the pool and the maximum size of the resource pool for each resource.

The shared portion of the resource pool only indicates that available resources can go to a specified resource pool or other resource pools. However, when resources are consumed, they go to one resource pool and are not shared. The shared part of the resource pool may improve resource utilization when there are no requests in a given resource pool (this means that the resources that were attributed to an empty resource pool can go to other resource pools).

The default resource pool can be modified but not dropped.

## Resource Pools (continued)




Resource pools can set the MIN and/or MAX for CPU and/or memory. The minimums for a resource across all pools cannot exceed 100 percent.

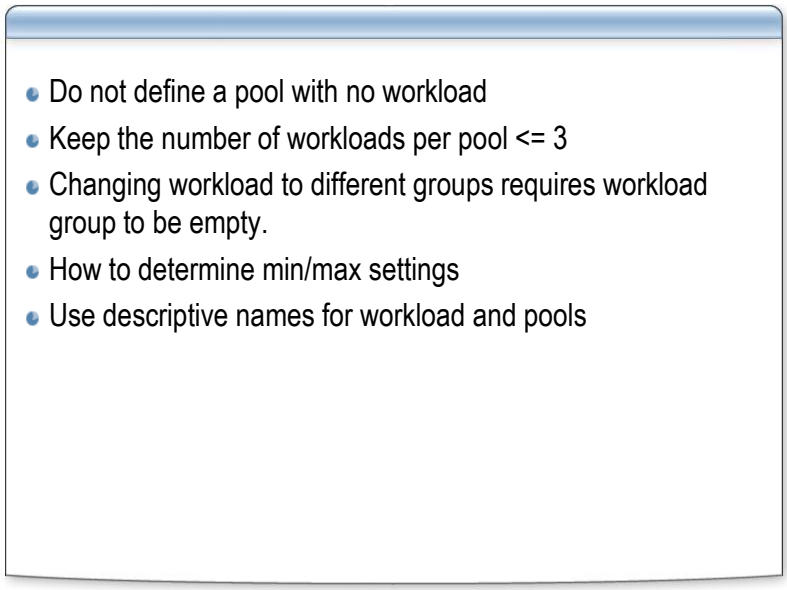
The effective maximum for a pool is calculated as the minimum of:

1. 100 percent or the sum of the minimums for all other pools.
2. The maximum setting for that pool.

For example, Pool 1 is set to a maximum of 80 percent. However, there are two other resource pools and the minimum for those resource pools adds up to 30 percent. So, Pool 1 has a maximum of 80 percent, or 100 percent minus 30 percent. Therefore, the effective maximum for Pool 1 is 70 percent.

You calculate the shared percentage by subtracting the minimum for the resource pool from the effective maximum. Therefore, if Pool 1 has a minimum of 20 percent, the shared percentage is 70 percent minus 20 percent, or 50 percent.

# Resource Pool Best Practices




You should define resource pools only if you have assigned them to a workload.

As soon as the resource pool is available at runtime, SQL Server 2008 applies the memory boundaries even if there is no workload group assigned to the pool. This is because SQL Server has to account for the resource pool and use the limits even when there are no sessions using the resource pool.

## Determining the workloads assigned to a resource pool

You can determine if there are any resource pools that do not have any workload groups assigned to them by using the following left outer join:

```
-- in metadata
Select p.pool_id, p.name, g.group_id, g.name
From sys.resource_governor_resource_pools p left outer join
Sys.resource_governor_workload_groups g on p.pool_id = g.pool_id
Order by p.pool_id
--In Memory
Select p.pool_id, p.name, g.group_id, g.name
From sys.dm_resource_governor_resource_pools p left outer join
Sys.dm_resource_governor_workload_groups g on p.pool_id = g.pool_id
Order by p.pool_id
```

## Determining the MIN and MAX settings

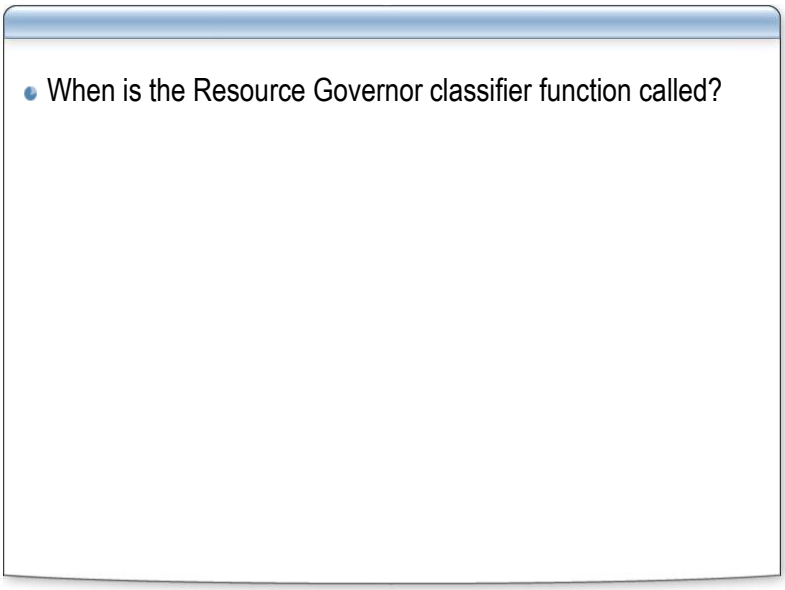
To determine the resources that a workload consumes, you first need to define the workloads and leave them assigned to the default resource pool. Then, you need to use the Performance Monitor (Perfmon) **SQL Server:Resource Pool Stats** and **SQL Server:Workload Group Stats** objects to answer the following questions:

| Question | Action to Take |
|---|---|
| What is the CPU requirement? | Determine a CPU bandwidth estimate by multiplying the average total CPU usage per request by the average number of requests per second.<br>Use the maximum, average, and standard deviation of CPU usage to determine whether a maximum CPU limit is necessary. |
| What is the number of concurrent requests? | Use the maximum and average number of concurrent requests statistics to determine the minimum concurrency factor. |
| What is the total memory required? | Use the maximum and average total memory used to determine the memory requirement. |
| How much memory is needed for a single query? | Use the maximum and average memory per query statistic to determine how much memory is needed. |
| Is any query not running due to lack of memory? | Use the average time waiting on memory queue statistic to determine whether any queries are blocked because of memory availability. |

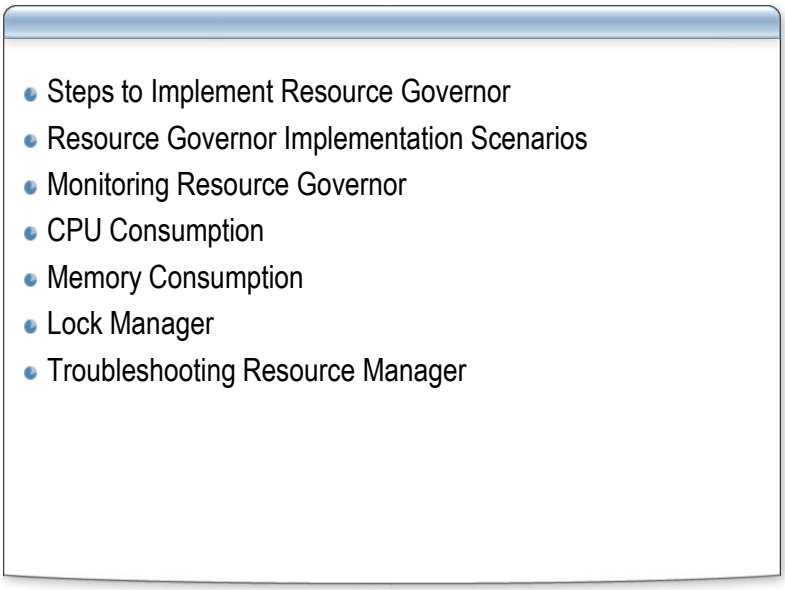
## Naming pools and workloads

All resource pool and workload group names are public facing. When creating resource pools and groups, you should choose names that do not disclose information about the nature of the applications that you are running on the server. For example, a workload group named *CompanyPayroll* provides an obvious indication of the nature and criticality of the applications that use the workload group.

# Section 1 Review



# Section 2: Implementing Resource Governor




## Introduction

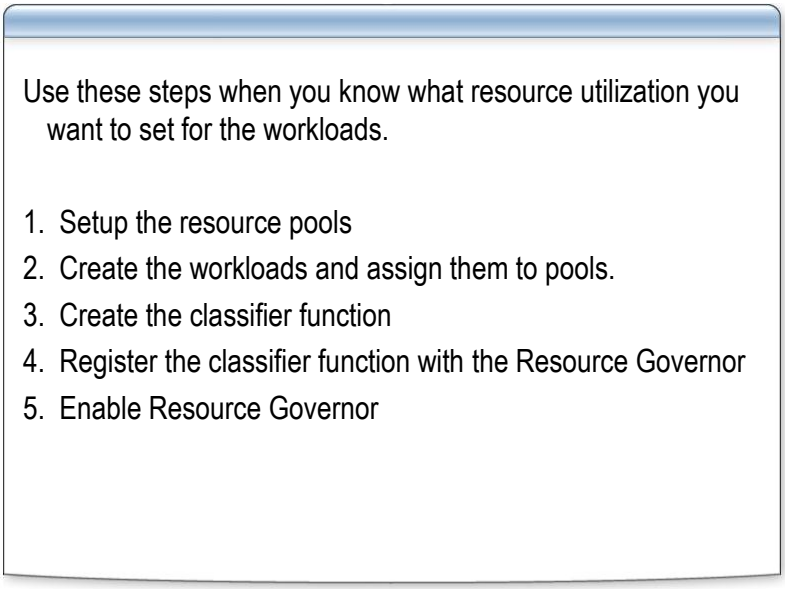
This section will provide the steps to implement Resource Governor. The steps depend on if the workload utilization is predefined or unknown. We will also discuss performance monitor counters available for Resource Governor.

## Objectives

After completing this section, you will be able to:

- List the steps for implementing Resource Governor.
- Explain several implementation scenarios for Resource Governor.
- Monitor Resource Governor statistics.
- Identify Resource Governor issues related to CPU consumption.
- Identify Resource Governor issues related to memory consumption.
- Explain how the lock manager works with Resource Governor to control the acquisition of row-level locks and to avoid lock memory errors.
- Troubleshoot the problems associated with Resource Governor.

# Implementing Resource Governor when workload utilization is unknown




There are two different approaches to implementing Resource Governor. The first approach can be used when you have identified different workloads but are unsure of the resource utilization of each workload. The second approach assumes that you have identified the resource utilization for each workload.

The following steps describe how to implement Resource Governor when resource utilization is known. For example, you want to prevent the administrators from running queries which take more than 10 percent of the server memory, and limit queries submitted by marketing from using more than 50 percent of the CPU.

1. Create additional and/or alter existing resource pools with the appropriate settings.

```
CREATE RESOURCE POOL poolAdmin
WITH (
         MAX_MEMORY_PERCENT = 10
)
;
create RESOURCE POOL PoolMarketingAdhoc

WITH (MAX_CPU_PERCENT = 50)
```

2. Create additional and/or alter existing workload group(s) with the appropriate settings, and assign each workload group to a specific resource pool.

```
create workload group groupAdmin
using poolAdmin
go
```

```
Create workload group groupMarketing
using PoolMarketingAdHoc
```

3. Create or alter a UDF to perform the classification (for example, a function returning the name of the workload group that each session will be assigned to, based on some connection information, such as login name or application name). Note that the UDF is also referred as the classifier function when configuring the Resource Governor.

```
use master
go
CREATE FUNCTION rgclassifier_v1() RETURNS SYSNAME
WITH SCHEMABINDING
AS
BEGIN
    DECLARE @grp_name AS SYSNAME
      IF (SUSER_NAME() = 'sa')
          SET @grp_name = 'groupAdmin'
 else if 'UserMarketing' = SUSER_SNAME()

              SET @grp_name = 'GroupMarketing';

      else
        set @grp_name = 'default'

       return @grp_name;

end
```

4. Register the classifiier function with Resource Governor as follows:

```
ALTER RESOURCE GOVERNOR WITH (CLASSIFIER_FUNCTION= dbo.rgclassifier_v1)
COMMIT TRAN
GO
```

5. Start the Resource Governor and apply all of the changes by running the following command:

```
ALTER RESOURCE GOVERNOR RECONFIGURE
```

To clear the Resource Governor statistics, use the following command:

```
ALTER RESOURCE GOVERNOR Reset Statistics
```

To unregister the classifier function, either use:

```
ALTER RESOURCE GOVERNOR DISABLE
```

Or use:

```
ALTER RESOURCE GOVERNOR WITH (Classifier_Function = NULL);
ALTER RESOURCE GOVERNOR RECONFIGURE
```

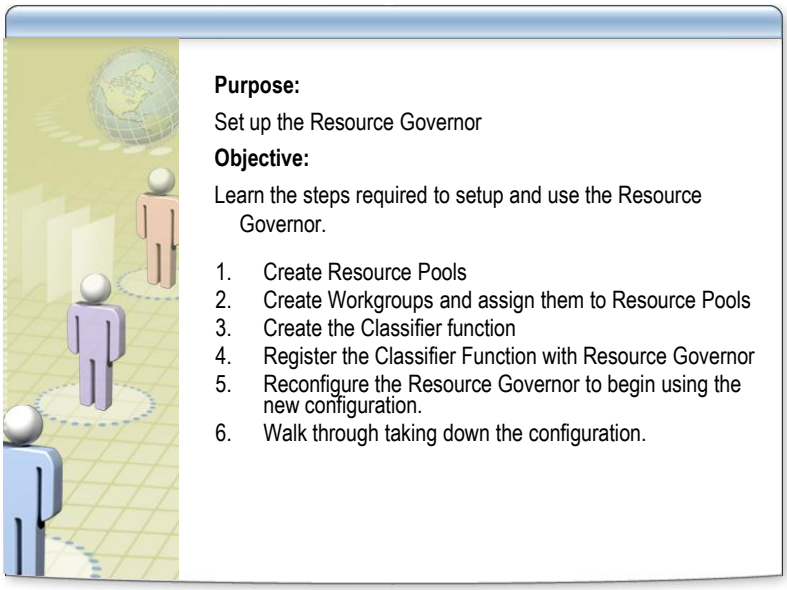
When you are unsure of the resource utilization of the workloads, the steps to implement Resource Governor are:

6. Create workload groups.

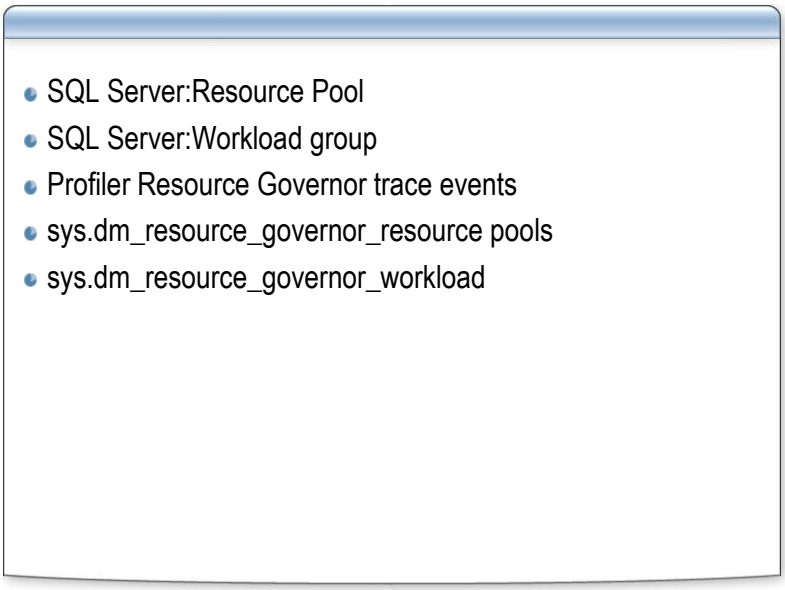
7. Create a function to classify requests into a workload group.

8. Register the classifier function in the previous step with Resource Governor.
9. Enable Resource Governor.
10. Monitor the resource consumption for each workload group by using the **SQLServer:Workload Group Stats** object in Perfmon. This object helps you monitor the amount of CPU and memory that each workload consumes.
11. Use the Performance Monitor logs to establish pools. Based on the statistics collected in the previous step, you can define minimums and maximums for resource utilization.
12. Assign the workload group to the pools.

# Demonstration 1: Implementing Resource Governor



# Monitoring Resource Governor




SQL Server:Resource Pool Stats Object contains performance counters for Resource Governor resource pools. The object contains CPU and memory counters for the resource pools. For more information see books online topic:

SQL Server, Resource Pool Stats Object

SQL Server:Workload contains performance counters for the Resource Governor workload groups. The object has counters for the number of requests and CPU and memory usage for the workload group. For more information see books online topic:

SQL Server, Workload Group Stats Object

You can use the following SQL Server Profiler events to monitor Resource Governor:

| Event Class | Description |
|---|---|
| CPU Threshold Exceeded | Indicates that Resource Governor detects a query that exceeds the CPU threshold specified for REQUEST_MAX_CPU_TIME_SEC |
| PreConnect:Starting | Indicates when a LOGON trigger or Resource Governor classifier function starts executing |
| PreConnect:Completed | Indicates when a LOGON trigger or Resource Governor classifier function finishes executing |

SQL Server Profiler also has a new column called *groupid*, which shows the resource pool that a session is assigned to. A value of 0 in this column indicates that the session is assigned to the internal resource pool, and a value of 2 indicates that it is assigned to the default resource pool.

User-defined resource pool IDs start at 256.

## Resetting Resource Governor statistics

You can reset statistics in the **sys.dm_resource_governor_resource_pools** and **sys.dm_resource_governor_workload** DMVs by running the following command:

```
ALTER RESOURCE GOVERNOR RESET STATISTICS
```

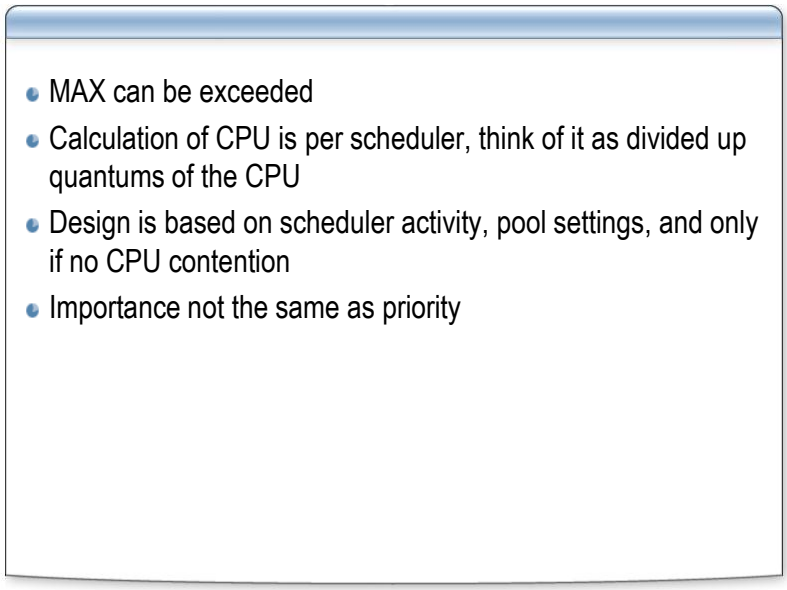
Running the command above resets the following counters for groups:

- statistics_start_time
- total_queued_request_count
- total_request_count
- total_cpu_usage_ms
- total_lock_wait_count
- total_lock_wait_time_ms
- total_query_optimization_count
- total_suboptimal_plan_generation_count
- total_reduced_memgrant_count
- max_request_grant_memory_kb
- blocked_task_count
- max_request_cpu_time_ms
- total_cpu_limit_violation_count

Running the command above resets the following counters for pools:

- statistics_start_time
- total_cpu_usage_ms
- out_of_memory_count

# CPU Consumption




## MAX CPU

If a pool has a maximum CPU usage of 50 percent, SQL Server can exceed that if no other activity is present. If the CPU has bandwidth, SQL Server still uses the available processors.

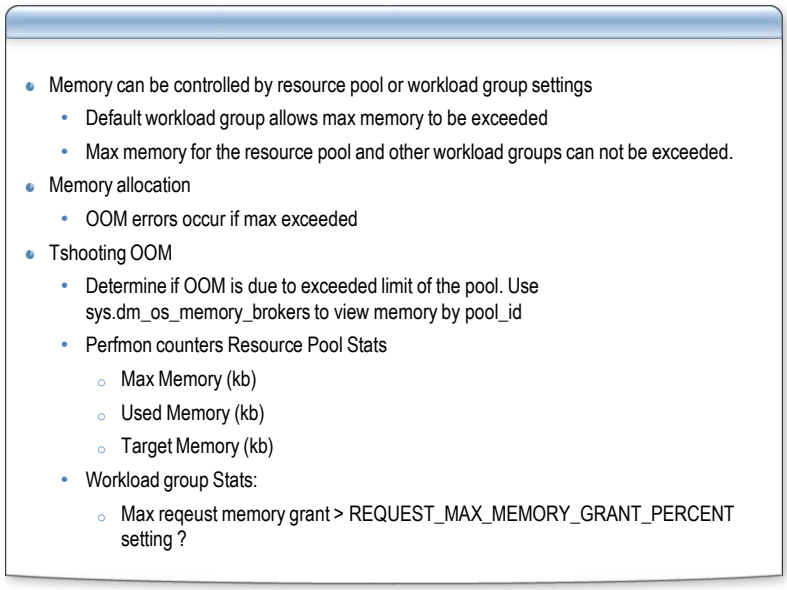
The MAX CPU limit becomes a hard limit only when exceeding this limit causes another pool to drop below its MAX setting. SQLOS always attempts to maximize the CPU resource usage. So if there is no CPU contention, each pool can exceed its MAX CPU setting because SQL Server always attempts to use all available resources.

> **Note:** Extended stored procedures, linked server activity, and some SQLCLR activity may not be controlled by Resource Governor. Therefore, if one resource pool uses distributed queries heavily, it may exceed its CPU usage for the resource pool.

## Group importance

When you create a workload group, you can assign the group an importance of LOW, NORMAL, or HIGH. Importance is not the same as setting the priority of a thread. The LOW, NORMAL, and HIGH settings translate to giving threads a weight rather than a priority. The weight is applied to threads within the same resource pool.

# Memory Consumption




## Controlling memory

You can control memory either by specifying the MIN and MAX options for resource pools or by specifying REQUEST_MAX_MEMORY_GRANT_PERCENT and REQUEST_MEMORY_GRANT_TIMEOUT_SEC for a workload group.

## Resource pool

The resource pool MAX option is an enforced limit, unlike the CPU MAX settings. The default workload group will allow a statement to use more memory than specified so that the statement succeeds. This behavior provides the same behavior as in previous editions of SQL Server. However, the memory manager cannot allow the resource pool MAX to be exceeded because it cannot guarantee that it could quickly trim itself. Therefore, the resource pool MAX for memory is a hard limit.

You will notice that this is different from Resource Governor *CPU Scheduling*, which allows extra CPU bandwidth to be used. The CPU can be easily re-adjusted during the next quantum to reduce its usage by a worker. If memory is distributed (above the MAX), SQL Server cannot guarantee that it can force the memory to be immediately returned. Therefore, the memory settings become hard limits.

## Memory allocation

Memory requests that cause the total target to be exceeded will fail.

This is important to how Resource Governor changes the default behavior of SQL Server. SQL Server can now fail with out-of-memory (OOM) errors even if plenty of memory is

still available. This is because the resource pool MAX limit has been reached and the allocation attempt is being rejected per the configured values.

If a memory request fails, memory from emergency memory can be used if the memory request is part of a critical section. For example, in order to perform a rollback, the thread needs additional memory. If this memory request causes the MAX memory to be exceeded, memory from emergency memory is used so that the rollback can continue.

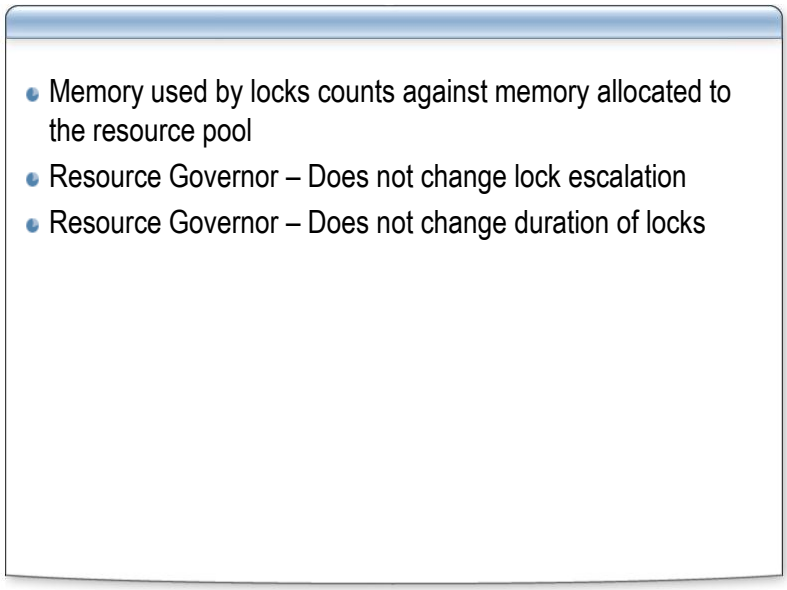
## Troubleshooting OOM messages

If OOM occurs, you should check how many workload groups are using the same resource pool. If you have more than one workload group using the pool, you should consider moving workload groups and create a new resource pool to be used by only one group.

To troubleshoot OOM:

- Query **sys.dm_os_memory_brokers** to check the relative memory distribution and trend inside the resource pool. Too many requests in a really small memory space can lead to an overloaded workload group or resource pool and cause OOM errors.
- Start Perfmon and collect data by using memory-related resource pool counters to get the target and current memory usage for memory grants, cached memory, and compile or optimizer memory. If the current values are greater than the target values, it means that the resource pool is overloaded. Also check Max Memory(kb), Target Memory(kb), and Used Memory(kb) counters for the pool. If Used Memory approaches the Max Memory, OOM may occur. In this case, you should consider changing the pool memory limits. In addition, you should check the **Workload Group Stats: max request memory grant** counter. If the counter value exceeds the value determined by the REQUEST_MAX_MEMORY_GRANT_PERCENT setting in the workload group, then the query will likely fail. So, you should consider changing the workload group limit.

## Lock Manager

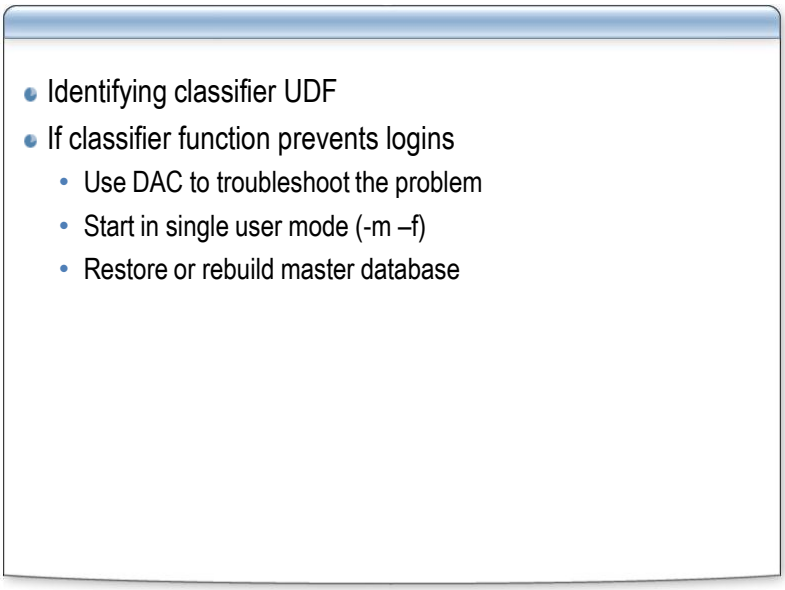



The SQLOS memory allocators use the resource pool ID to support the configured total limit so that a new allocation will simply fail when a lock block or lock request allocation exceeds the total limit of the current pool.

To understand this better, assume that you have set Pool A to a memory setting of 5 percent. If you start a query in Pool A and it escalates to a table lock as soon as 60 percent of the 5 percent of visible memory is reached, it would not take much memory to escalate to a table-level lock. For example, you have a 2 gigabyte (GB) computer that has 1.5 GB of total visible memory available for the SQL Server instance. On this computer, 60 percent of 1.5 GB is 922 megabyte (MB) but 5 percent of 1.5 GB is 75 MB, and 60 percent of 75 MB is 45 MB. If Pool A escalates to a table lock each time its lock memory reaches 45 MB, the concurrency for other pools may be severely impacted.

To avoid this situation, Resource Governor retains system-wide escalation thresholds. This means that the queries in Pool A will not escalate until the overall lock manager targets are reached. The side effect for Pool A is that it may continue to acquire row-level locks and get *out-of-lock memory* errors.

# Troubleshooting Resource Governor




## Identifying the classifier UDF

You can use the following queries to determine which classifier UDF is used by Resource Governor:

```
use master
go
-- Get the stored metadata.
select object_schema_name(classifier_function_id) as 'metadata Schema',
object_name(classifier_function_id) as 'metadata UDF'
from sys.resource_governor_configuration
go
-- Get the in-memory configuration.
Select object_schema_name(classifier_function_id) as 'Active schema',
object_name(classifier_function_id) as 'Active UDF name'
from sys.dm_resource_governor_configuration
go
```

You can also use the following SQL Server Profiler events to identify the classifier UDF:

- **PreConnect:Starting event class**. Provides the ID and name of the classifier UDF.
- **PreConnect:Completed event class**. Provides the ID and name of the classifier UDF.

## Classifier UDF prevents logins

If the classifier UDF is preventing logins from succeeding, you can use the following options to troubleshoot the problem:

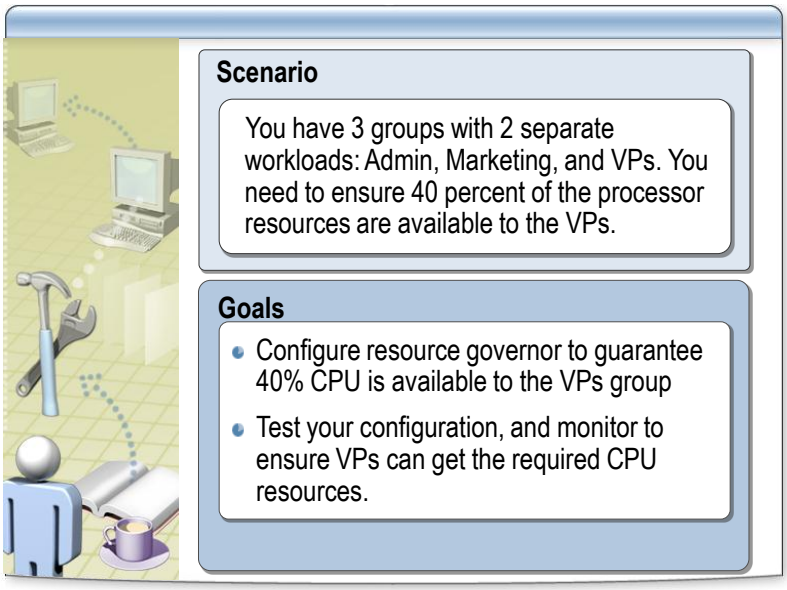
- Login by using DAC to troubleshoot the problem.
- Start in single-user mode by using the sqlservr –m –f command. This command bypasses classification and enables you to use the internal resource pool.
- Restore or rebuild the master database. If you do this, you must have scripts for the classifier UDF and the pools, or else you will not be able to re-create them.

# Section 2 Review

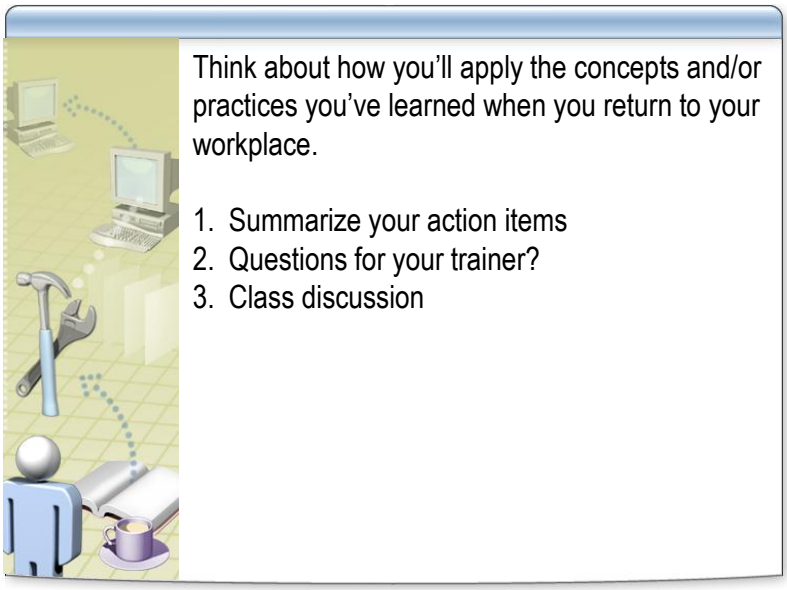
- Can the maximum CPU usage for a resource pool be exceeded?
- Are limits for memory considered hard limits or can they be exceeded?
- How can you make a connection to SQL Server if the classifier function used by the Resource Governor is causing connection timeouts?

21

# Lab 1: Setup Resource Governor for Workloads



# Action Planning Exercise




When you return to your workplace, you'll want to use the concepts and practices you've learned to positively impact your IT environment. In this exercise you'll think about what you've learned and create action items that you can follow up on when you return to your workplace.

## Step 1: Summarize Your Action Items (5 Minutes)

How can you apply what you've learned to your workplace?

| | |
|---|---|
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |

## Step 2: Questions for your trainer? (5 Minutes)

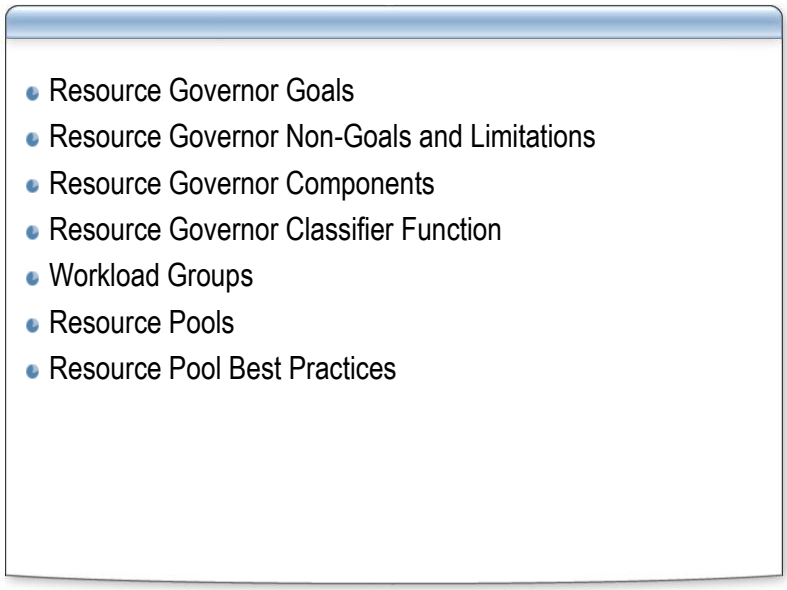
Record any questions you have about accomplishing the Action Items you listed above. The trainer will allow time for discussion before moving to the next module.

| | |
|---|---|
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |

## Step 3: Class Discussion

Notes:

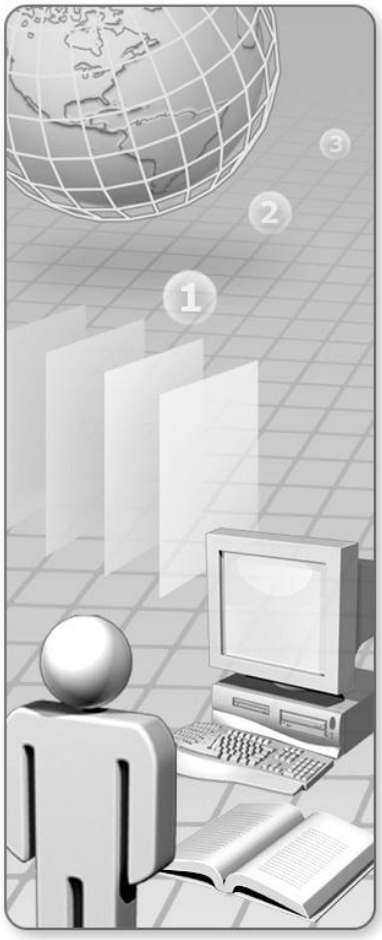
# Module Summary




In this module we learned how to implement the Resource Governor. The Resource Governor is useful when you have different workloads using the same SQL Server instance. In order to implement the Resource Governor, you have to create a classifier function, workload groups, and resource pools. Resource Governor allows you to specify how CPU and memory are consumed by different workloads.

Microsoft | Services

# Outgoing Assessment



# Outgoing Assessment

We are nearing the end of this Workshop*PLUS* course, and it's time for the Outgoing Assessment (You of course remember the Incoming Assessment!).

For a refresher on why the Incoming Assessment and Outgoing Assessments are so important – and what the benefits are to you, to your management, and to Microsoft – turn to the "Incoming Assessment" section near the front of this Workbook. In case you don't look at the "Incoming Assessment" section now, we want to be sure that you are clear about the benefits to you of the Outgoing Assessment.

### Benefits to you:

You get an opportunity to see how much you've learned – a measure of improvement.

- Students are not always aware of how much they've learned.
- Students are happily surprised – even amazed – at how much they learn and how much their scores improve.

You finish the workshop feeling really good because:

- You know that your hard work was worth it.
- You feel more confident than ever in your ability to perform well on the job.

The subject matter experts who created this assessment believe that it covers the key points that all students should learn from this workshop. After the Outgoing Assessment, the Trainer will review each question and answer, making sure that you understand all the key concepts.

> **Note**: Your results are anonymous. Your Assessment results are never associated with your name.

### Privacy / Anonymity

We want to again mention the steps being taken to the ensure privacy and anonymity of your results:

- During the workshop, your Assessment answers and results are only associated with your personal identifier or your student number (depending on the workshop). In all cases, the answers and scores are never associated with your name.
- Sometime after the workshop, the scores from the class will be entered into a database. The only data entered is similar to "Student 1 answered question 3 as C." Your real name is never associated with your student number.
- No one will see or have access to your individual Assessment scores – not your manager, not others in your company, and not any Microsoft-employed Technical Account Managers, Engagement Managers, or Support Professionals.
- Only aggregated class-average results might be shared with your management.