SYBASE
An SAP Company

TECHNICAL WHITE PAPER

Managing Workloads with ASE: Techniques for OLTP Scaling and Performance Management with Large SMP and Shared Disk Clusters

Contents:

List of Figures:

Principal Author

Jeff Tallman jeff.tallman@sybase.com

With special thanks to the various ASE engineers and PSE's who answered customer related questions that in turn contributed to this paper - and especially Dave Putz for his meticulous review and both he and David Wein for their detailed explanations and corrections.

Revision History

Version 1.0 - November 2010

Introduction

Over recent years, hardware trends, business performance requirements, and the separation of applications from data have combined to push many applications from small, dedicated environments able to perform predictably and consistently to larger SMP, virtualized or clustered environments. This has in no small part been driven by such trends as:

- Server consolidations as increases in CPU speeds and multi-core processes enable multiple separate environments to be combined.
- Increases in multi-core processors and chip threading technologies has translated event the smallest machines into large SMP monsters with 10's to 100's of virtual CPU's.
- Breakdown of data silos as business applications need access to multiple data repositories.
- Decreasing staff positions due to budget realities has driven server consolidations in an attempt to regain control of the operational complexity of managing the widely scattered dedicated systems and exploding power and cooling costs.
- The promise of server consolidations in reducing both the latency of data movement and complexity of data federation within the datacenter – providing the business with the desired real-time, cross spectrum data access capabilities
- The promise that commodity hardware-based clusters could achieve the same performance scalability as last year's super-sized SMP machines at a fraction of the cost but with increased availability

What is "large" SMP? A decade ago, one might have said anything above 8-10 CPU's. In fact, when ASE 12.5 was introduced in 2000, one of the features – asynchronous log service (ALS) – was aimed at large SMP with the caveat that it would not help with systems unless they had more than four engines. In practice, it was much more than four engines – mostly it didn't provide any benefit until in the 12-16 engines or larger range. With today's Intel XEON or AMD Opteron based machines with four quad, 6, 8 (Intel) or 12 (AMD) core CPU's, a single small workgroup size server supports 16-48 cores of processing power at a very low price (<$30K US). On the higher end of the x86 line, both HP and Sun offer 8 socket systems that would support 64 cores in a single host at well under $100K US. Chip multi-threading has increased this - entry-level XEONS now supporting 2 threads per core (32-64 virtual CPU's), and at the extreme end, quad socket Sun Niagara T2 based systems support 32 cores and 8 threads/core for a total of 256 virtual CPU's. Adding to the complexity on one end is shared disk clusters and the strong push for horizontal scaling – meanwhile on the other end, server virtualization software allows allocations of fractional CPU's as virtual CPU's in LPAR's, containers or other hardware partitioning. The result is that a single system may legitimately support dozens to hundreds of virtual CPU's or appear that way when viewed from the collective of all the hardware partitions. So, does that mean yesterday's large SMP is now only mid-range with "large" SMP now being 16-30 cores?? And do we have "very large" SMP and "extremely large" SMP for the 30-60 and 60-100+ range systems?? What is "large"?

The reality is that "large" probably starts in the 12-16 engine range but is more of a function of the mix of workloads on the system competing for the 10's of resources than an actual quantity. The problem is that many DBA's are suddenly facing for the first time "large" SMP environments even on "small" hardware as well as facing multiple different and competing application requirements within that environment. To make matters worse, hardware sales often oversold multi-threaded CPU scalability with few system administrators realizing that their brand new shiny 256 virtual processor pizza box still only had the CPU horse power of the refrigerator rack it replaced – and wasn't the Cray replacement envisioned. The result sometimes has been chaos. Many times, shortly after a server consolidation goes live, a new hardware platform with new virtual CPU technologies, or a new suite of applications is added to an existing server that appears to have plenty of capacity – performance plummets. As a result, the business impact is immediate and measurable in the millions of dollars. Unfortunately as few DBA's are prepared for this, the problems persist for weeks, escalations spiral ever higher within the IT, business and vendor organizations – IT staff lose sleep and the business suffers customer dissatisfaction as well as the mounting business losses. It all could have been prevented.

The problem was that what was forgotten was that this is client-server computing. While system administrators eagerly adopted hardware and OS virtualization to provide server consolidation of multiple applications – they forgot a critical part: a good chunk of the workload for applications (and a good chunk of the logic) happens within the DBMS. Unfortunately, the typical DBMS system either inherited multiple applications in a consolidation attempt or simply grew up that way – the end result was often a system sustaining 1,000's of users and dozens of applications….and no control.

The goal of this paper is to introduce to readers how ASE and ASE/Cluster Edition features can be employed to prevent such problems from even starting – or to resolve them once panic begins. While the techniques are mainly aimed at the larger SMP environments, they can apply to small or mid-range SMP systems as well – depending on the application workloads involved. While those features are key, it is also crucial to realize how different tuning problems can affect ASE scalability – tuning problems often caused by leaving parameters that control ASE internals at their default – or misconfiguring them due to lack of understanding. As a result, the first major portion of this paper will discuss ASE internals tuning, followed by cache tuning, and then the workload management features. As a result, there are three main sections to this paper:

Drivers & Trends – Brief synopsis of CPU, OS and disk technologies changes that either are driving a much more critical look at workload management or are aiding scalability (but require re-tuning ASE to benefit).

System Task Configuration & Tuning – A detailed look at common configuration issues for system tasks that often inhibit scalability.

Virtualization & Workload Management – A thorough discussion of ASE features to aid in virtualizing applications within ASE as well as managing the workloads.

One aspect of scalability not considered in this document is application design/implementation, with exception where key use of a feature requires application design considerations (such as with transaction log durability options). It doesn't matter how well you tune the environment,

segment the application modules, etc. if the application design isn't scalable. A bad application design – even in one area – can cripple scalability – and nothing – hardware, OS, DBMS – can do anything about it. You need to fix the application. It is crucial to realize that before the techniques discussed in this paper can provide any benefit, the application either has to be first eliminated as a scalability bottleneck or the location of the bottlenecks understood so that these techniques can be used to contain the problem. Some of the more common application scalability bottlenecks include:

- Sequence generation tables (e.g. update table set key=key+1)
- Unpartitioned heap tables
- Unpartitioned monotonically increasing indexes (e.g. trade date)
- Row-wise processing or atomic single row operations (vs. small sets)
- Repeatedly creating/dropping temp tables vs. using connection lifespan temp tables
- Lock contention due to application design in addition to contention inherent in the some of the above design implementations

The last one - lock contention - frequently is one of the biggest scalability bottlenecks. Unfortunately, it is almost always application related and consequently impossible to address outside of application design techniques - a topic out of scope for this paper. Even beyond locking, some of these are hard to avoid (e.g. monotonically increasing indexes) and while they may be contributing to the issues, they are not major concerns. Others such as sequence generation tables simply don't scale at all – even in small SMP environments.

While this paper does discuss the Sybase ASE internals to a particular level, the examples used may be simplified or may be simply an example of the typical algorithm but not the specific algorithm used by Sybase. This is deliberate to avoid requiring non-disclosure agreements with readers. Additionally, in places the paper may state "<capability> is not currently supported" in ASE. This is not intended to infer there are future plans to support such a capability, although it is possible that such is the case.

The Impact of Hardware Trends

As we all have been aware, over the past 10 years, there has been a substantial shift in how processing capacity of systems was provided. Prior to 2000, the main thrust for increasing processing speeds and capacity was largely a processor speed race as CPU clock speed and overall chip transistor densities were the main factors. At the time, there was a huge gap between RISC and CISC (x86) based processors as the reduced instruction set of the RISC processors allowed greater speed due to fewer clock cycles per instruction. Further restricting capacity, the only widely accepted OS for x86 systems prior to 2000 was Microsoft Windows – an OS generally thought to be too unstable or too insecure for enterprise systems.

In the early 2000's, this all began changing. First, Linux hit its stride and became more mainstream. Secondly, the CISC based x86 processor family adopted more RISC style execution in the P4 series Pentiums with most instructions requiring only a single clock cycle. The result was incredible pressure on traditional UNIX vendors as commodity systems were easily able to outperform most of their mid-range systems at a fraction of the cost. For example, the original single core P4 Pentium in a dual CPU configuration could achieve nearly the same throughput as a Sun SPARC v880 with 8 UltraSPARC III processors. At the same time, CPU engineers were realizing that the traditional means of improving processing power was rapidly dimensioning compared to effort expended, and application developers realized that traditional methods of scaling applications using increasingly larger SMP systems was becoming cost prohibitive in the post-tech bubble burst. This set the stage for two technology advances we all are familiar with today: multi-core/multi-threaded CPU's; and horizontal scaling with clusters. While a lot of attention has been focused on CPU's and scaling, two other technologies actually solve even bigger bottlenecks with respect to IO's – SSD's vs. disk IO's; and Network EtherChannel (network bonding) vs. network IO's. Each of these four technologies are discussed in the following sections.

Multi-Core & Multi-Threaded CPU's

It is hard to say which really arrived first – chip multi-threading (CMT) via Intel's P4 XEON Hyper Threading or multi-core with AMD's Opteron. What is undeniable is that the multi-core design of the Opteron was proven to be the most effective at first and for a while was the main focus of increasing processing capacity. Today, this technology is so common place that most home-user commodity systems support quad-core processors and 6 or 8 core commodity processors are now on the market.

While multi-core processors were a slam-dunk, chip level multi-threading was less so. The initial hyper-threading implementation from Intel did not provide the readily apparent performance boost that users were expecting – and consequently quickly disappeared from the scene. However, as chip die sizes started putting practical physical limits on the number of cores, hardware engineers began looking at chip multi-threading as a means of improving CPU utilization. To date, there really are only a few different types of chip multi-threading designs: Fine-Grained Multi-Threading, Coarse-Grained Multi-Threading and Simultaneous Multi-Threading (SMT).

Strands

Before we look at the different threading techniques, a bit of terminology is useful. From the hardware's perspective, the multiple execution threads are referred to as "strands" as it is the "strand" that holds the hardware state for the different processing threads. There is a key difference in realizing that a thread represents a parallel execution module that may or may not run simultaneously vs. a "strand" represents the hardware state that depending on the implementation *may* allow the threads to execute simultaneously. Hardware state for a process is best thought of as the collection of hardware registers, instruction pipelines, caches and transaction look-aside buffers (TLB)'s used to execute a process. To implement "strands" some of these resources need to be duplicated, shared or halved. Obviously, not every resource can be duplicated or else the result would be another processing core. Consequently, if a CPU without strands enabled is showing 75% utilization, enabling strands at the best will gain an additional 25% utilization – even though it will appear to the OS as an additional virtual CPU and perhaps misleading a DBA into thinking they have double the capacity as previously available.

To understand what hardware threading really is all about, you first have to understand the effects of instruction pipelining and parallel pipelining with today's CPU's. Oftentimes, today's program instructions are broken down into multiple processor instructions. Some of these instructions can be executed in parallel while others need to be executed sequentially. To really simplify things, take the following section of pseudo-code:

```
C = B + A
D = C*E
```

This actually breaks down to a number of operations – such as retrieving the values of A & B from memory, saving the value of C to memory, performing the integer/float operations, etc. In some cases, such as retrieving the values of A & B from memory, those instructions can be issued in parallel as they are independent. However, the next instruction (math addition) has to wait until those are complete. Similarly, the next statement has to wait until the previous operation was complete prior to beginning as it is dependent on the value for C. In the early efforts to introduce parallelism within CPU's this notion of instruction parallelism was known as Instruction Level Parallelism (ILP) – and significant effort was spent in determining which instructions could be executed out of order to increase the parallelism and efficiency in processing. For example, in the above, the instruction to fetch 'E' from memory could be issued in parallel with the instruction to fetch 'B' and 'A'.

While ILP worked to some extent, it still left a lot of the processor underutilized which was perceived as an issue as system throughput goals were ever increasing. This brought about the more recent Thread Level Parallelism (TLP) focus of increasing processor utilization. As a bit of background to the thinking, remember, back at the operating system level, there are always more processes needing access to the CPU than available CPU's, consequently the operating system implemented a time-sharing or time-slicing mechanism in which different processes would run for an allotted period of time. With parallel pipeline execution this could be represented with a picture similar to the following in which the solid blocks represent an instruction being executed within one of the CPU's pipeline stages:

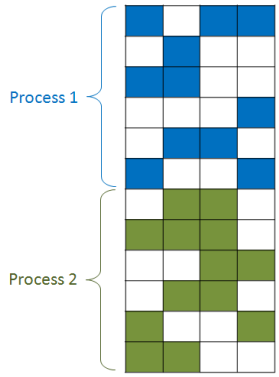


Figure 1 – CPU parallel pipelines & process execution

If you look at the above, you can see that most of the time, the CPU is not well utilized and runs at 50% or less real utilization. The primary basis for threading is two-fold. The first is that many instructions involve fetching from main memory (i.e. reading a page in the 100GB of data cache) and accessing main memory outside the on-chip L1 & L2 caches can take several clock cycles to simply even fetch the data. As a result, there is a lot of "blocked" execution time in which a process running on a CPU really isn't running –but rather waiting for a main memory access to return. Some of the first attempts at threading such as in IBM's Star processors implemented Hardware Threading (HWT) by simply context switching the tasks whenever a blocking instruction such as a main memory access was hit on the current thread of execution – called 'switch on event'. The second basis for threading is that some instructions are dependent on others to complete before performing. In a lot of those cases, the dependent instructions often have little parallelism within the same stage, consequently there is a lot of idle resources even when execution is occurring. Consequently, with enough duplication of internal registers, etc., a single processing core could actually execute simultaneous threads.

The goal of CPU threading was to add enough hardware resources to allow more than one process to execute on the same chip without having to reload the registers during the context switching or similar problems. By doing so, time spent idling while waiting for a main memory access would be filled as would be the idle resources due to instruction dependency. Ideally, this would end up with a CPU/process execution picture similar to the following:

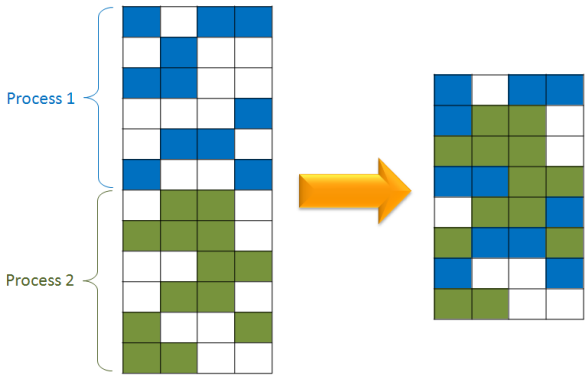


Figure 2 – CPU parallel pipelines & parallel thread execution

The key is CPU *utilization* – *NOT* application performance. This is another misunderstood aspect. A single process application's performance will not be improved by using CPU strands – and in fact is often degraded. First, depending on the type of chip multi-threading

implemented, the processing core may have to spend some amount of time performing the context switch between the threads. However, the biggest impact is that to the extent that shared or halved resources restrict the complete simultaneous execution, each of the threads loses some processing time (blocked) while the other thread accesses the shared resources. The chief benefit is that in an un-stranded system, a single process may dominate the processor resulting in lower CPU access by other applications. When CPU strands are enabled, other processes may run at the same time frame by leveraging the CPU idle cycles – however since only a single process can actually be executing at a single point in time, the original application's CPU utilization decreases. This results in longer response times and performance degradation for the original application – but ***faster*** response times for other applications for a total gain in system CPU utilization.

Consequently, if trying to increase the scalability of a single application – such as a DBMS – configuring multiple strands per processing core will help if the amount of concurrency is fairly high and CPU intensive – but at the cost of decreasing the performance of a single task. The common effect is that online transaction processing benefits while monolithic batch jobs suffer mightily. As a consequence, batch processes or long transactions often may need to be redesigned to leverage parallel DBMS tasks to avoid exceeding the batch window or transaction execution response time expectations.

Another key aspect is that strands are a hardware configuration and not an OS kernel setting. Consequently, the DBA will not be able to tune or adjust the number of strands per core without help from the system administrator. On some systems, the strands can be adjusted fairly dynamically via OS utilities such as Sun Solaris's Lights Out Manager or IBM AIX's SMiT or even OS commands such as psradm – while on others (such as Intel x86), it requires a hardware BIOS setting configuration change.

Fine-Grained Multi-Threading

Fine-Grained Multi-Threading is based on round robin task switching on a CPU cycle basis. Sometimes this is referred to as Time Multiplexed Threading (TMT) as at its crudest implementation, each thread is simply allocated an equal time share of the CPU – which is more close to Vertical Threading than anything else. A simplistic characterization would be that rather than a pure 'switch on event' of Vertical Threading, fine grained threading almost presumes that a lot of blocking instructions will be issued in today's large memory systems and that simply time slicing the threads will likely coincide with this fairly often and provide better hardware utilization. It also was easy to implement and could be done at lower power allowing the CPU to run cooler (no doubt a factor in Sun's marketing of this as "Cool Threads").

As also could be easily envisioned – it is the technique that has the most undesirable performance impact on a single process. If a single processing core implemented four strands, each thread would receive 25% of the total processing time available. This also would not be very efficient as the real problem that threading was supposed to resolve was CPU utilization. Low CPU utilization was often a direct result of the CPU having to wait for a multi-cycle event such as a cache miss (requiring main memory access), a code branch (requiring instruction fetching) or FPU access. The goal of multi-threading was to allow other threads to execute during these long multi-cycle idle waiting periods. However, if a strict TMT implementation was used – even if the time slice was measured in a microseconds – with each thread executing for

a set period, there would be no overlap of execution. The hope would be that a thread's time slice would expire at the point a multi-cycle instruction was invoked, and that instead of the CPU waiting for its return before continuing, it would simply work on something else. When it became time to process that thread again, by then, the multi-cycle event would have completed and execution could begin immediately. However, for a DBMS that uses a large "data cache" in main memory, the number of main memory fetches would be fairly high – especially, the more random the memory accesses such as when traversing an index tree or accessing different objects vs. scanning through the rows on a single data page.

Because the efficiency would only improve if a multi-cycle event happened right at the edge of a timeslice, the efficiency gain of TMT is best suited for large numbers of threads that are computational and memory driven. While it might be thought that reducing the time-slice could help, the problem is that the context switching time would increase – most likely offsetting any gain after a certain point. As a result, fine grained multi-threaded CPU's have to duplicate a lot of the hardware resources to hold each thread's state so that the time to perform a context switch is as minimal as absolutely possible. Generally speaking, very few (other than Sun's Niagara T1 & T2) CPU's implement true TMT as the primary threading mechanism – although Sun refers to their threading implementation using the more generic Chip Multi-Threading (CMT) vs. the more specific TMT it implemented. However, many CPU's implement some form of TMT in combination with VMT or SMT as a way of detecting/avoiding deadlocks on shared CPU resources.

The most common fine-grained multi-threaded CPU is perhaps the Sun Niagara T1 & T2 processors (TI fabrication). The Niagara threading module switches between available threads *each clock cycle* – almost as if TMT was implemented at every clock tick – based on an LRU method of thread selection. The theory of fine-grained multi-threading is based on the fact that the popular method of processing involves a multi-stage instruction pipeline involving common stages such as fetch, decode, execute, memory access, write-back. Since most early processing cores could only execute one instruction per cycle, the goal would be to have the different strands at different stages of the pipeline – consequently, one strand would be at the execute stage while another might be idle in the memory access stage due to a main memory fetch. However, Sun's Niagara improves on this model slightly by marking a thread (or strand) unavailable if the thread is waiting on main memory access, instruction branch or similar event. The result is a more efficient TMT implementation in which threads waiting on a multi-cycle event are not checked (or executed) until the event completes and the strand is re-enabled. Additionally, in the T2 series, Sun implemented a multi-threaded pipeline, reducing the some of the pipeline conflicts and pipeline latency. If you consider the multi-stage pipeline mentioned earlier, a perfect system would be if each stage of the pipeline required the same amount of time and the number of strands were equal to the number of stages. However, in reality, some of the stages are fast (e.g. execution or cache access = 1 cycle) while other stages are slower (e.g. memory access = multiple cycles if cache miss). The result is that some of the threads execution time not only now still has the memory access latency that was always there, it also now has "pipeline latency" in which it is waiting on the next stage of the pipeline. The net effect is that a single thread of execution is actually slower on a fine-grained threaded core than on a single stranded (or non-threaded) core. This effect has been seen repeatedly by Sybase customers that have adopted Sun Niagara T-series based systems as a dramatic drop in overall system throughput. The reason is that each ASE engine represents a single thread to the

processor and each engine is only getting 1/4th (4 threads per core) or 1/8th (8 threads per core) of the CPU access due to the threading and some of that time is spent waiting on pipeline access.

Coarse-Grained Multi-Threading

Sometimes coarse-grained multi-threading is called "switch on event" or "vertical multi-threading". Rather than running multiple threads in an overlapping fashion as with fine-grained threading, coarse-grained threaded processors only run a single thread at a time. However, when that thread invokes an instruction that requires multiple cycles, the processor performs a thread context switch to another task. Unlike in fine-grained architectures where a context switch is effectively happening every cycle and therefore a context switch must be 0 cycles, a coarse-grained threaded processor can use a few cycles to perform context switching and therefore be of a simpler design from fewer hardware resource duplication. Of course, part of the problem is that when the original thread's instruction returns, it has to wait for the now executing thread to hit a context switching point such as a cache miss. This is where the TMT aspects enter. To avoid a single task from monopolizing the CPU when no cache miss happens, a coarse-grained multi-threaded systems typically also implement a timeslice for any thread and invoke a context switch at the end of the timeslice expiration regardless.

The most common coarse-grained multi-threaded processor available today is the Sun SPARC VI (Fujitsu fabrication) which it calls Vertical Multi-Threading (VMT).

Simultaneous Multi-Threading (SMT)

In simultaneous multi-threaded processors, a super-scalar pipeline is used that allows more than one instruction to be issued at the same time. As noted in the diagram above, different processes (or threads from the same process) could execute simultaneously if the registers and other hardware is duplicated. However, as with VMT threaded processors, SMT threaded processors also implement some of the same 'switch on event' logic which means that today's SMT threaded processors really combine the features of VMT threading with simultaneous thread execution. The most common SMT threaded processors are the IBM Power P-series and Sun SPARC VII (Fujitsu fabrication) – that latter being interesting as it shows the natural progression from VMT threading in the SPARC VI to the SMT threading in the SPARC VII. In addition, Intel has also adopted SMT-like behavior for both the Itanium 2 (IA-64) as well as the re-birth of their HT implementation for the XEON Nehalem cores. The result is that predominantly non-Sun Niagara systems will likely be mainly SMT based vs. fine-grained with a few coarse-grained VMT systems from Sun SPARC VI series also in market.

Recommendations for ASE

As illustrated so clearly by the Sun Niagara T-Series systems, unless the application workload can be evenly balanced across all the strands, performance using chip multi-threading can suffer significant degradation. As a result, there are a few key recommendations for running ASE on multi-threaded processors

Number of strands ≈ number of engines

Unless the host machine is being shared with other CPU intensive applications or between multiple ASE instances, the total number of strands (and hence virtual CPU's) enabled on all the cores should be approximately the same as the maximum number of engines anticipated to be used by ASE. For example, if running ASE on a Sun T5440 (4 sockets, each chip with 8 cores), if planning on only running 32 engines, the cores should only have a single strand enabled (as if non-threaded). However, if you anticipate 48 engines, you should enable only 2 strands per core for a total of 64 virtual CPU's. This also holds true for SMT-based systems. Sun SPARC VII based systems support 4 strands per core – but if planning on running an M5000 with 4 quad core SPARC VII's (16 cores) with ASE configured for only 24 engines, you should only enable 2 strands per core (32 total virtual CPU's).

Note that if at the edge – for example 32 engines and 32 virtual CPU's – it may help to enable an extra strand per core to provide CPU time to OS services. This could be especially true for OS's like Solaris that implement file system IO using POSIX threads which can execute across any CPU. Consequently, if using a buffered file system for tempdb or file system DIRECTIO, having the additional virtual CPU's might help. Similarly, on AIX systems, asynchronous IO is often handled by queue engines – consequently running 32 virtual cpu's to support 16 engines on a 16 core P-series host may provide better overall throughput. Although, as with anything, configurations should be tested and then monitored closely in production as each application's workload may impact the system differently. Note that in each case – no matter what the threading implementation – running more than one strand per core may reduce the response time for a single process but increase overall throughput. As a consequence, benchmark configuration should include user concurrency as a prime metric vs. single user (or batch process) response times.

Processor sets/domains and ASE Engines

An unfortunate aspect of chip multi-threading and virtual CPU's is ensuring optimal performance by avoiding unnecessary hardware cache transfers and system bus activity. Commonly, in a dual stranded (two threads per core) system, each pair of subsequent strands are on the same processor core – for example, strand 0 & 1 on core 0, strand 2 & 3 on core 1, etc. What happens beyond that largely depends on the processor architecture. Some cores share a pool of L2 cache while others have a dedicated amount of L2 cache per core or subset of cores. Unfortunately, the OS is typically not aware of these details. As a result, during normal task management, it will schedule a task on the next virtual CPU that is available. The problem is that as a result, an ASE engine may bounce from one core to another – or even from one CPU to another – as the OS schedules the engine on the available virtual CPU's. For example, if an 8 engine ASE is running on a system with 64 virtual CPU's (8 cores with 8 strands per core), all 8 engines initially may be scheduled on the first 8 strands – all on the same core. While the effect is marginally better than starting 8 engines on a single processor host – it is only marginal. Worse, during the next OS task scheduling cycle, the engines may be scheduled on the next 8 strands on the second core. If the L2 cache is not shared between the cores, than an L2 cache flush needs to occur. The same happens as the OS scheduling moves tasks across physical processor chips vs. just across cores within the same processor chip.

Some OS's have the capability to specify which virtual CPU's are members of a processor set or other hardware partitioning such as domains/LPAR's. How you define these processor sets or hardware partitions will largely depend on the amount of resources required by the various applications sharing the resources. While it may appear obvious that processor sets can be used to keep an ASE engine from being rescheduled from strand 0 core 0 to strand 10 (core 2) the opposite may also be true. We may deliberately want to ensure that engines are executing on separate cores for maximum CPU access. For example, if a 16 engine ASE is to share an 8 processor/16 core/64 strand host with a mix of other non-critical applications or OS resources, constraining all 16 engines to 4 physical cores with 4 strands per core is not likely to achieve the same DBMS response time as if each engine was one strand on a separate core.

While such manual balancing of cores & virtual CPU's may not be necessary if the host is a dual stranded, dedicated DBMS host for a single server instance, the more strands per core and the more different types of applications hosted, the more critical such tuning will need to be. This is especially true when considering engine groups as will be discussed later.

ASE Tuning: Runnable Process Search Count

Traditionally, out of the box, ASE is tuned more for a single-stranded processor than multi-threaded ones. Each engine effectively operates as a mini-operating system. As a result, it tends to hold the CPU as long as it can before yielding back to the operating system. Some of this "holding" of the CPU is time spent simply looking for work – whether polling for new network activity (such as new queries submitted) or IO completions. If ASE is sharing the host machine with other applications or other ASE instances, reducing the 'runnable process search count' from the default of 2000 to 100 or 200 may help the performance as ASE will spend less CPU cycles 'spinning' looking for work and yield the OS to the CPU faster. This may help in multi-strand environments as a process yield of the CPU could allow more CPU time to be allocated to another strand that needs the time.

The key factor in determining if the '*runnable process search count*' can be safely reduced is to monitor the MDA table monSysWaits during normal processing (vs. peak). Wait event ID 179, 'waiting while no network read or write is required' reflects this runnable process search as the interpretation is that the server is simply spending time looking for work (either a incoming network request or a result to send back) yet has nothing to do. The key is to determine if this a consideration is to look at the top 10 Wait Event causes from monSysWaits. If WaitEventID=179 is in the top 10 and within an order of magnitude or is larger than the others in the top 10, the runnable process search count likely can be reduced. Consider the following example:

WaitEventID	Waits	WaitTime	Description
29	1399734780	4929606	wait for buffer read to complete
215	**1127548743**	**27844254**	**waiting on run queue after sleep**
35	670307183	4153181	wait for buffer validation to complete
179	**335988317**	**10035324**	**waiting while no network read or write is required**
250	324655266	167672101	waiting for incoming network data
124	256994610	6105113	wait for someone else to finish reading in mass
209	75463669	340471	waiting for a pipe buffer to read
251	62546271	344880	waiting for network send to complete
41	58470129	3473384	wait to acquire latch
31	32361806	36401	wait for buffer write to complete
214	**19911597**	**1403956**	**waiting on run queue after yield**
150	18083160	23776944	waiting for semaphore
52	11842516	48193	waiting for disk write to complete
51	9703708	27945	waiting for disk write to complete
55	8071811	5609	waiting for disk write to complete
36	4774219	20192	wait for mass to stop changing

```
272     3886481      134998 waiting for lock on PLC
 54     2438135       30805 waiting for disk write to complete
```

The above system is experiencing primarily IO performance related issues as the top 3 wait events (by number of waits) are all related to physical reads – or could be attributed to physical reads (i.e. WaitEventID=215). After that gets resolved – perhaps by improving the index selection for queries – the second biggest wait is the engine idle loop looking for more work. However, reducing the runnable process search count should wait until the IO problems are resolved. While an engine will not yield to the OS if it has outstanding IO's (so the runnable process search count has no impact on outstanding IOs), once the IOs return, the engine may be put to sleep prior to the task submitting subsequent IOs – resulting in slower response times.

Looking at the above example, if the physical reads are resolved, the CPU contention is actually fairly low. The amount of time waiting for CPU access after sleeping due to lock/physical IO/network send (WaitEventID=215) is much less than 1ms (27,844,254 ms/1,127,548,743 Waits = 0.02ms/Wait). CPU wait time for yielding due to exceeding time slice (runnable state – WaitEventID=214) is also less than 1ms (1,403,956 ms/19,911,597 Waits = 0.07ms/Wait). As a result, the runnable process search count could probably be reduced by at least a factor of 10 and likely the wait times would still likely be less than 1ms. A second way of monitoring this would be to take several snapshots of sp_who and look at the number of 'runnable' processes versus sleeping or running. "Runnable" processes are those that are simply waiting on the CPU and represent the "run queue" for the ASE engines. If the run queue is less than 5 per engine (averaged), again, quite likely the runnable process search count could be reduced. Perhaps the best way is to simply look at monSysLoad if using ASE/CE 15.0.x or ASE 15.5.

```
create existing table monSysLoad (
        StatisticID                     smallint,
        InstanceID                      tinyint,
        EngineNumber                    smallint,
        Sample                          real,
        SteadyState                     real,
        Avg_1min                        real,
        Avg_5min                        real,
        Avg_15min                       real,
        Peak                            real,
        Max_1min                        real,
        Max_5min                        real,
        Max_15min                       real,
        Peak_Time                       datetime,
        Max_1min_Time                   datetime,
        Max_5min_Time                   datetime,
        Max_15min_Time                  datetime,
        Statistic                       varchar(30) NULL,
)
external procedure
at "@SERVER@...$monSysLoad"
go
grant select on monSysLoad to public
go
```

While monSysLoad was added for monitoring ASE Cluster Edition, in ASE 15.5, some of the CE focused MDA tables work even in a non-Cluster Edition Server. Consider the following snapshot of data from an idle ASE 15.5 (non-Cluster Edition) server:

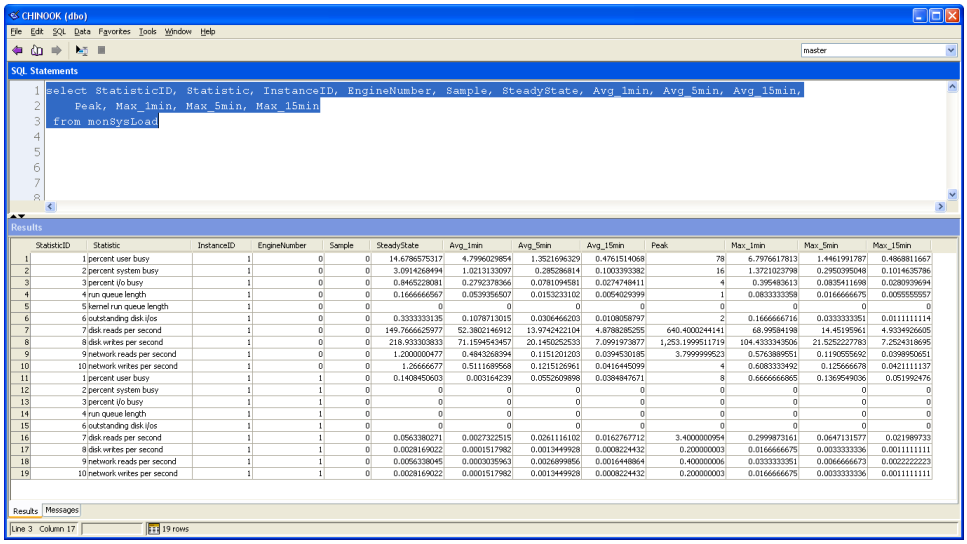


Figure 3 – Sample monSysLoad output

The run queue length in the above never peaked above 1 for engine 0, and engine 1 had no run queue length at all. Of course this was an idle server, but by watching 5 & 15 minute averages you can get a very good idea of how busy the server is and whether CPU can be given back to the OS without impacting application response times.

ASE Tuning: Max Online Engines & Increasing Strands

That last trick in the previous paragraph is probably a good way to consider when you might need to increase the number of strands and the number of engines. If the ASE run queue starts exceeding about 5-10 per engine, you probably should consider bring on additional engines. Of course, this depends on the type of application. We used to use the rule that if all the engines are greater than 90% utilized, but with today's systems, most of the database will reside completely in memory. For example, the average ASE customer has a database between 120-250GB in size. Even typical mid-range SMP hardware platforms are shipping today with 128GB of RAM, which allows most of the database to reside in memory. The problem with this is that it artificially increases CPU utilization. For example, an in-memory table scan does not need any physical IO – so unless there is contention for a lock or a network send happens, the SPID doing the tablescan will not sleep – it will simply cycle on and off the engine for a full timeslice. If the tablescan is unavoidable (i.e. DSS query or small table), then with even a little concurrency, the engines would quickly hit 90%+. However, if looking at the run queue, you see a pattern of always having 10 or so SPIDs in a runnable state or you see the WaitTimes from MDA showing >1-2ms per wait, ASE might benefit from having the additional engines online. This may require increasing the number of strands first – then increasing the number of engines.

The key point is that the engines have to be kept busy – this hasn't changed in the days since the 90%+ utilization rule was first discussed.

Commodity Scale-Out Clustering

The other hardware trend that had captured a lot of attention in the past 5 years has been the concept of horizontal scaling – "scale-out" – using commodity or mid-range SMP in place of large SMP platforms. While some of this was started using Solaris and AIX boxes, it really never

gathered much steam until Linux systems began supporting clustered environments as the attractiveness of using multiple cheap boxes to replace aging but phenomenally expensive large SMP hosts drove a lot of attention. Much like CPU-threading has its share of misconceptions, scaling out using multiple smaller machines in place of a single larger machine also has a number of misconceptions. The biggest of these misconceptions is that "scaling-out" can achieve the same results as "scaling-up" (larger SMP) from an application context. While scaling-out may be able to approach the same theoretical capacity for a single application, a lot of workload management issues need to be considered for most applications to achieve modest scalability – without which clustering will result in performance degradation (much like CPU threading can experience). However, there are several general types and considerations for clustering.

Logical/Virtual Clusters

A logical or virtual cluster leverages multiple independent instances that are unaware of each other (no cluster-ware) to provide scalability. Synchronization between components is achieved at a logical level using software by duplicating the components. While the most common implementation of logical clusters is app server or web server farms, DBMS's have supported logical clusters using bi-directional data replication for decades. Logical clusters from a DBMS perspective are especially useful in geographically distributed environments as it reduces the WAN latency of centralized implementations. Since a logical cluster is virtual, it can be built using other supported clustered systems as components. A key trend in this direction is the aspect of internal clouds and cloud computing overall as the location transparency aspects of cloud computing won't be realized if network latency for geographically distributed users is apparent.

Companion Clusters (HA)

Companion clusters are strictly a high-availability solution typically involving one or more primary nodes and one or more failover nodes controlled by OS level cluster services. There are several key differences between this and shared disk clusters, including:

- A database is only mounted on one instance at a time. Any read or write activity has to be directed to that node either by connecting to it or having operations re-directed to it by middle-ware or cluster services.
- If a failover happens, the cluster management software instructs the failover node to first "take over" the hardware resources, assume the network personality (e.g. logical hostname, etc.), and either restart or recover the databases in an already running instance.
- Workload management under normal operations is managed at the cluster service level consisting of one or more applications. Moving a workload from one node of the cluster to another typically results in an outage. For example, if more than one database instances are running on one node of the cluster, one of them could be moved to another node by relocating the cluster service. However, this involves stopping the service and restarting on the other instance

The key aspect to companion clusters is the notion of relocatable services – typically involving restarting the service on a different physical host. One or more application services exist within a logical host comprising all the hardware resources and network personality – and these logical hosts can be relocated among multiple physical hosts either due to a failure or to manage workloads. In this context, many of today's grid implementations that heavily rely on virtual machines essentially are merely companion clusters.

Sybase first started supporting companion clusters with VAX/VMS clustering in the late 4.x days in the early and mid 1990's. The technology later was resurrected with ASE 12.0's HA support. However, it is strictly a 1+1 (1 primary and 1 failover) high availability solution that relies heavily on OS cluster services and support. Today, a number of customers choose instead to run ASE in virtual machines for workload management reasons – which depending on the capabilities of the VM software in use – may not provide as transparent of a service relocation as with the ASE HA option.

Shared Disk Clusters (SDC)

Shared disk clusters have largely replaced companion clusters from an availability perspective as most virtual machine deployments are strictly for workload management. Additionally, shared disk clusters have been largely viewed as a method of providing application scalability – especially using commodity hardware.

The key difference between a shared disk cluster and companion cluster is that a shared disk cluster presents a single system image across the cluster nodes – irrespective of which node you connect to – where a companion cluster merely provides host transparency. In order to provide this single system image, shared disk clusters have to provide several enabling technologies:

- Cluster storage management – all the storage is accessible in read/write mode from any host with the exception of private storage at each node. Storage may be managed external to the DBMS using cluster file systems or within the DBMS as shared storage. A key requirement is the notion of cluster locking within the file system or DBMS to prevent multiple nodes from attempting to write to the same disk blocks at the same time.
- IO fencing – the storage subsystem can isolate portions of the cluster to prevent logical data corruption in the event that the cluster suffers from a split-brain incident in which one or more nodes loose synchronization with the other nodes of the cluster. Typically, this is implemented via the SCSI-3 Persistent Group Reservation (PGR) capability within the SAN controller. SCSI-3 PGR implements IO fencing by blocking processes from outside a list of known processes from writing to the disks. The cluster software controller (DBMS) is responsible for maintaining this list.
- Cache synchronization – the single system image extends to the in-memory data cached within the DBMS. This is critical to prevent logical corruption as modifications to old versions of a page in one node could have the effect of either undoing or corrupting the on-disk image of the same data page.

- Workload management – in order to provide any aspect of scalability, the shared disk cluster needs to provide workload management features such as application routing, load balancing and failover services.

Sybase provides shared disk cluster support via ASE Cluster Edition. Note that shared disk clustering and companion clustering typically are mutually exclusive – especially when using virtual machines.

Shared-Nothing Clusters (MPP)

Shared nothing clusters are based on the premise of a divide & conquer strategy for large complex job processing. Simply put, the workload from a single job is considered so over-whelming on a single host that it is divided into multiple logical units of work that are dispatched to other nodes in the cluster and then re-merged for the final results. For DBMS systems, this typically revolves around the physical and logical IO processing – as the volume of data –either already in memory or still on disk – requires too much IO bandwidth for a single machine or more commonly too much CPU time evaluating/processing the data pages to see which rows qualify. This strategy for cluster relies heavily on techniques such as:

- Distributed query processing – a single query can be split into multiple query fragments and each fragment executed independently with results merged into a final single result.
- Data partitioning – data can be effectively partitioned across the nodes with each node associated with a slice of data. This is crucial to maintain data cache effectiveness as focusing query fragments after the same data partitions to the same nodes reduces cache volatility.
- Federated system image – a logical transaction that spans data partitions can be implemented as a distributed transaction to avoid the cache synchronization issues with only single instance updates. Based on the data partitioning association, this can be made transparent to the application as the data partition association would route the respective transaction commands to the appropriate node.

While in the past, Sybase MPP (nee Navigation Server) provided an MPP solution, Sybase ASE currently does not support an MPP implementation. The rationale is that MPP is largely a DSS oriented solution as the necessity of breaking up large queries to be executed in parallel is the key.

Solid State Disks (SSD)

While solid state disks have been present for decades, only in recent years have they become cheap enough and the capacities large enough that they are viable for most systems – large or small – and should be considered a key element in the scalability considerations for DBMS systems. While this seems obvious, most IT professionals are not aware of how much an impact SSD's can have on their system's scalability (or they discount it due to misconceptions about SAN caching) or understand the configuration options and best implementation strategies.

Due to rotational latency, today's fastest tier 1 hard drives are only capable of 200 IOs per second (IOPS) average and 420 IOPS peak (outer-edge). Using a RAID stripe across 5 devices raises that maximum to barely 1,000-2,000 IOPS – assuming no overhead of the RAID implementation. The problem is that today's transaction velocities are such that most high performance OLTP systems are striving to achieve high 10's of thousands of IOPS – both from a disk write as well as disk read perspective. For example, a financial system seeking to achieve 70,000 inserts a second and a health care system that runs queries that require 600,000 IOs to complete both have similar issues with normal hard drives – they are just too slow. This also has dramatically driven up storage costs as the number of devices necessary to achieve even acceptable performance results in multiple refrigerator sized racks in the data center. By contrast, a single SSD can sustain between ~10,000 to ~200,000 IOs per second and a 5 disk stripe would provide 300GB of storage – sufficient for most databases – and take up less than 6 inches of space.

What might not be quite as obvious is the impact on scalability. For example, DBMS transaction logs or journals are often written in a serial fashion using very small IO's to reduce the system response time. The problem is that such logs are fairly small in size and often implemented on a single disk stripe – or due to the serial nature of use – only affect a single disk stripe at a time. This significantly reduces the throughput to what can be achieved with a single disk stripe. While SAN cache can buffer this problem, sustained activity eventually will overflow the SAN cache. This is especially true if other high volume write activity (e.g. tempdb) is also using the SAN cache. On the read angle, because the most efficient join method for fully indexed joins involves nested loop joins, DBMS read activity tends to be extremely random compared to storage. As a result, SAN read-ahead caching algorithms are fairly ineffective at pre-fetching data from disk ahead of the query requests. As a result, queries that require physical IOs truly are hitting the disk with each IO request – and the slower a query runs, the more opportunity for contention with other users.

SSD Types & Configuration

Interestingly, some companies object to the term SSD applied to their products today. The rationale likely stems from the fact that a decade ago, SSD's were primarily RAM disks with battery backup along with a normal hard disk for recovery purposes should the battery fail. While such devices provided improved performance over a normal hard disk, in order for the hard disk to provide recovery, the RAM disk was effectively limited to the total throughput leveling for sustained writes to the hard drive. Today's SSD's are completely different in design – and are based on flash memory – no battery backup is required – and no hard drive necessary for recovery. Companies that object to the term SSD typically refer to their products as 'flash drives'. For the purposes of this paper, we will use the term SSD to refer to flash drives.

Today, most IT professionals are aware of solid state disks due the rapid commoditization into the home PC market. However, enterprise level solid state disks are slightly different from the PC versions. PC versions are generally based on Multi-Level Cell (MLC) NAND flash memory chips and as a result, suffer a write penalty for high volume random writes characteristic of DBMS activity. Additionally, due to the design, MLC based drives have a much shorter life span for high volume writes – although this can be extended significantly through on-drive cache and driver supported trim function. Enterprise solid state disks are base on Single Level Cell (SLC)

NAND chips. For example, the Intel X-25M and OCZ Agility series are more consumer oriented MLC implementations while the Intel X-25E and OCZ Agility EX series are SLC implementations. Even more confusing is that Fusion-IO markets a SLC ioDrive and a MLC ioDrive in the same family (ioDrive Duo) but different capacities (and obviously different specifications).

Beyond the basic types of SSD's, there are some configuration considerations as to where the SSD is located with respect to the host machine(s).

SAN-based SSD's

These typically are the most expensive, but often can leverage SAN infrastructure such as multi-hosting, SAN management tools as well as SAN redundancy capabilities such as disk block replication. One unique option for SAN SSD's is the capability to on-the-fly relocate LUNs (or sub-LUNs) experiencing heavy IO to SSD and then age it out. For example, EMC's Symmetrix V-Max supports flash based SSD's in the SAN using a Fibre Channel interface.

SSD-based Storage Array

An SSD based storage array is very similar to a SAN in that the storage array controllers are located in the storage array itself. Typically, the same type of HBA that is used to attach to a SAN can be used to attach to a storage array – although vendor specific firmware may be needed to optimize for an SSD-based storage array. The storage array typically can be partitioned into one or more storage domains which support access to a limited number of hosts – which differs from the nearly limitless capability of a SAN. For example, a Sun F5100 can support a maximum of 4 domains. Redundancy at the device level is supplied by standard array RAID levels (not to mention fewer moving parts). Redundancy at the HBA level or array level can be achieved by using host based mirroring – usually via host OS LVM soft RAID implementations – that can mirror the writes to one HBA to a second HBA and different storage array. Surprisingly, this is cheaper than you might think. Current web list price as of May 2010 is ~$46,000 US for 480GB (240GB if mirrored). With support for 4 storage domains – if mirrored, a single F5100 could support 2 different hosts with a single 120GB mirrored flash domain for ~$23K each. Two F5100's with 480GB each would support 4 hosts with a mirrored 120GB flash drive at the same cost ($23K each) but would be mirroring across storage arrays vs. within the same array. This could provide some extra redundancy by locating one storage array in a different room from the other.

Backplane Based SSD

These SSD's are actually on add-in cards that fit into PCI or other backplane based expansion slots within the host machine itself. Typically they have the highest throughput and fastest response times due to operating at the bus speed vs. the IO controller limitations. However, they are fairly expensive – often in the $10,000 US range per device – but don't suffer as from wear-leveling as SSD disks do. Often they are NVRAM based as are other SSD's, but some are DRAM based with battery to preserve memory state. Note that DRAM based systems suffer none of the NVRAM wear leveling issues and are much faster – but also more expensive per GB and less common as a result.

An unfortunate aspect of PCI based implementations is that they are often limited to one or two per host due to the limitation of requiring a backplane slot. Redundancy is provided by

using a form of RAID to stripe across the memory chips using an N+1 as well as substantially more ECC bits than a typical harddrive based RAID mechanism would employ. The result is a much more reliable device than a standard RAID-based disk based solution. In addition, similar to the SSD storage arrays, OS level mirroring can provide redundancy by mirroring the writes to a second backplane based SSD in the same host machine. Support, however, is currently limited to Windows and Linux operating systems for direct PCIe devices.

The primary suppliers of PCIe based SSD's are FusionIO™ and Texas Memory Systems (TMS). FusionIO™ supports a variety of ioDrive™ devices including the ioDrive™ and ioDriveDuo™ with 160GB and 320GB maximum capacities in SLC NAND implementations (MLC capacities are double) with throughput nearly 200,000 IOPS. TMS supplies similar devices via their RamSan™ 10 and 20 model drives with capacities up to 450GB.

Texas Memory Systems took this one step further with their RamSan™ 4xx/5xx/6xx systems such as the RamSan-620. The RamSan-620 encloses up to 20 256GB (5TB total) PCIe based SSD cards in a single 2U rack mount chassis that supports standard FibreChannel connections from up to 8 different host machines. The RamSan-6200 is essentially 20 RamSan-620's with a total of 100TB of SSD storage available via FibreChannel connections. Since the RamSan disk is exposed as LUN's to the OS, it may provide the ability to extend beyond the Linux and Windows limitation of the PCIe support.

SAS/SATA RAID-SSD's

This is probably the cheapest solution (at roughly $500/device) and the least reliable – but still much more reliable than internal disks. While the most common is SATA2 based, the SAS specification supports SATA2 compatibility, consequently a host-based SAS RAID controller should be able to be able to support multiple SATA2 SSD devices. Linux systems obviously can use either SAS or SATA2. However, these typically are the slowest. For example, Intel's X-25E supports 35,000 random read IOPS and 3,000 random write IOPS in capacities of 32GB and 64GB. Note that a high-end disk drive barely reaches the 200-400 IOPS range, so calling these slow is not exactly accurate – although they are slower compared to the earlier SSD configurations. For redundancy, you essentially construct a normal array using the internal disks. For example, you could implement 2 RAID controllers and multiple SSD devices per controller using RAID. Then similar to the others, you use OS LVM host mirroring to mirror between the two RAID controllers. If any single device fails, the RAID protects from complete failure. If a RAID card fails, the other RAID card and mirrored volume is still available. Support for this implementation is open to any OS/HW vendor that supplies a SAS or SATA2 RAID controller and would likely be less than $10,000US per host.

Recommendations for ASE

SAN based devices make sense for user databases that need to either share storage (such as ASE/CE) or need the availability provided via SAN-based disk replication. Internal backplane or SAS/SATA2 SSD's can be used as private storage for data that does not need to be shared – such as tempdb's – or when the availability architecture uses a logical implementation such as Sybase Replication Server.

Note that using SSDs may require OS tuning as well as DBMS tuning to gain faster speeds. Some OS's such as Sun Solaris, for example, have kernel configurations that control how quickly the OS will poll for completed asynchronous IOs (vxio:vol_default_iodelay for Solaris – discussed later). Reducing this from the default value to a substantially lower setting has often been suggested for years for SAN storage systems – simply due to the write caching within the SAN. SSD's will also benefit from reducing such configuration values. Additionally, ASE has several configuration parameters such as '*i/o polling process count*', '*i/o batch size*' as well as '*disk i/o structures*', '*max async i/os per engine*' and '*max async i/os per server*' that may need to be tuned as well. Unfortunately, the ASE configurations are based at a server level vs. a device level, consequently extremely high parameter values may require some caution (although most systems are grossly undertuned).

Optionally, you may wish to consider using dbcc tune(deviochar,<vdevno>,"<size>") to increase the i/o batch size issued to SSD's used for tempdb even without sp_sysmon highlighting it as a problem. Remember, the i/o batch size is generally only used by a process when the spid is performing a lot of write activity. Examples include checkpoint (flushing dirty data pages based on recovery interval), housekeeper (flushing dirty data pages), select/into, and bcp (bulkcopy) operations. For tempdb, the checkpoint flush and housekeeper should not be considerations (rationale discussed later), however, select/into is a definite consideration as well as bcp a possible consideration (if tempdb is used to stage in bcp'd data).

The following are best uses for SSD's to provide scalability within ASE. While the order they are discussed is likely the order in which preference should be made for scalability purposes, this may change due to site specific performance requirements. Additionally, other Sybase products in the infrastructure may benefit from using SSD's, such as RS stable devices, Sybase IQ temp space, etc. Consequently, you may need to balance the gains from ASE vs. allocating some of the usage to other products in your infrastructure.

Transaction Logs

One of the most commonly cited scalability issues with ASE is the transaction log semaphore. While the log semaphore can be an issue in very large SMP environments with very small transactions (a topic discussed later), more often the real problem is the log device – the log semaphore is just the visible symptom. . In order to assure full ACID properties of a transaction, any DBMS that uses write-ahead logging for durability (D in ACID) has to ensure that the writes to the transaction log actually are on disk prior to allowing the transaction to commit. Again, in order to provide full ACID, transactions are serialized to ensure consistency (C in ACID) during recovery. For ASE, this serialization is enforced by only letting one process write to the transaction log at a time. As a consequence, the process writing to the transaction log holds the log semaphore the entire time it is waiting for the physical IO to complete. The faster the IO completion – the quicker it releases the log semaphore. On SAN based storage, a physical write under normal load should take only 1-2ms on average. Unfortunately, this limits the system to sustaining only 500 (2 ms/IO) to 1,000 (1 ms/IO) transactions per second. A SSD device – even mirrored – would easily exceed this.

Tempdb

The logical assumption is to use an SSD for tempdb. While this does dramatically improve ASE startup times when tempdb is cleared, before deciding to placing tempdb on an SSD, the

system should be monitored for physical IOs against tempdb. In recent years – especially with ASE 15 – much of the physical IOs that took place in tempdb have been eliminated. In addition, if the data cache was configured such that tempdb was largely cached as is possible with today's larger memory systems, even more IOs can be avoided. As a result, on many of today's OLTP systems, tempdb on an SSD may not have the same payback as putting more volatile data on it.

However, while ASE has taken great strides in reducing the amount of physical IO's that happen in tempdb, unfortunately, a lot more physical IO's can occur than desirable. One advantage of using an SSD for tempdb is a lot of the normal I/O throughput tricks such as allocating in small chunks, /tmpfs or using SAN write cache is eliminated. Additionally, since typically tempdb's are non-recoverable, it is a slam dunk to use internal (backplane or SAS/SATA) SSD's vs. SAN based SSD's – which also helps by reducing the write cache impacts and IO throughput load of having tempdb on SAN based disks. The only caution about this is that for ASE Cluster Edition, we are specifically speaking of the local tempdb's using the internal SSD's as private devices – not global tempdb's, which need to be created on shared storage in the SAN.

Volatile Table/Indexes

One of the issues that can block scalability is volatile tables & index writes. Even though this is typically an asynchronous operation from the context of a single spid as the writes to data cache do not wait for physical IO to occur – the writes to data cache can be blocked by physical IOs on the pages being performed by the housekeeper, checkpoint process, wash marker, etc. There are some synchronous IO operations - such as when the spid needs to modify an index key that results in index tree maintenance/rebalancing. However, most data page modifications should be asynchronous via the housekeeper, checkpoint or wash marker.

The key to this is realizing that ASE uses the concept of a MASS (Memory Address Space Segment). A MASS essentially is a pool sized group of buffers used to synchronize IO to disk. For example, in a 2K server, each 1MB 16K pool in the default data cache will consist of ~64 MASSES of 16KB each. When large IO is performed (read or write), the large IO will use the entire MASS. Most disk IO operations these days are done via DMA access – which means reads/writes directly happen from memory. As a result, the memory (buffer) representing the pages being written or read from disk need to be protected from another concurrent process attempting to modify the same pages – or read a page that has only been partially populated from disk. ASE uses a MASS bit on each MASS as a type of mutex – when a MASS is experiencing a physical IO, the process that initiated the MASS holds the bit until the IO completes – thus protecting it from other users. Note that the entire MASS may not be written to disk. If only one page within the MASS is dirty, ASE will only schedule a single page IO. However, if several of the pages are dirty, it is more efficient for ASE to write the entire mass as a single IO vs. scheduling multiple IO's – one per page.

The point is that while physical writes are being done by the housekeeper, checkpoint process or synchronous IO operations, attempts by other processes to modify those pages will be blocked. This doesn't have to be just in the case of updates. If a table is experiencing heavy inserts at the end of the table (ascending inserts), the housekeeper may still try to flush the pages to disk as well as index tree rebalancing as the table grows The faster these writes occur, the faster the process can release its exclusive locks (or MASS bit for checkpoint, etc.) as well as

reduce the overall spid processing time. This can be monitored using monSysWaits or monProcessWaits MDA tables.

Unfortunately, as big as SSD's are these days, the databases seem to be bigger. Consequently, the best way to ensure optimum use for select tables/indexes is to create a segment that points to the device and then drop the default segment from it. Then you need to either drop and recreate the objects using the 'on segment' clause or use sp_placeobject followed by optionally a reorg to move existing data.

```
exec sp_addsegment ssd_segment, <db_name>, <ssd_device_name>
go
exec sp_dropsegment 'default', <db_name>, <ssd_device_name>
exec sp_dropsegment 'system', <db_name>, <ssd_device_name>
go
exec sp_placeobject 'ssd_segment', <tablename>
exec sp_placeobject 'ssd_segment', <tablename.indexname>
go
```

The types of tables to consider include those that are used for:

- Workflow queues, call center queues, etc.
- Sequential key generation tables (i.e. next key tables)
- Key business transaction tables (e.g. trades, clinical systems processing, etc.)

One that might appear to be an obvious choice is transaction history tables. While these do sustain a high write activity, the size often precludes using them as they would likely fill the SSD quickly. Unfortunately, the current implementation of sp_placeobject is not partition-aware, as a result, you would not be able to include the SSD in a tiered storage scheme without extensive manual intervention such as bcp'ing out the partition data.

Network EtherChannel

While SSD's have gained a lot of attention due to the orders of magnitude performance difference over their legacy counterparts, EtherChannel (aka network bonding) is providing another improvement in what has often become the slowest part of a server – the network. While it is true that disk drives are slow – most enterprise systems have been using SAN technologies for decades – and with the SAN implementation, gigabytes of cache and faster RAID processors to mask the slow speed of individual disks. The result is the most disk I/O's take less than 2ms per I/O even at small sizes such as 2KB pages when doing random I/O's. Coupled with I/O concurrency, the disk I/O throughput is often nothing short of astounding. But one of the biggest factors in forcing the network to be the slowest component is the simple fact that efficient data caching often eliminates physical I/O's or reduces it to just the writes which are cached by the SAN. Consider the following sp_sysmon fragments from a large (50+ engine) system:

```
                            per sec      per xact       count  % of total
-------------------------  ------------  ------------  ----------  ----------
Total Requested Disk I/Os        4159.9           1.5     7541885
Total Completed I/Os             4160.0           1.5     7542156

Total Network I/O Requests      12268.6           4.3    22242947       n/a
Total TDS Packets Rec'd          7048.5           2.5    12778964
Total TDS Packets Sent           7963.7           2.8    14438219

Total Bytes Rec'd             1415060.2           9.5  2565504145
Total Bytes Sent              1958430.1          13.2  3550633835
```

As you can see, the number of network requests is nearly 3x the number of disk I/O's. Meanwhile, the aggregated bytes sent/received per second is 3,373,490 – if we assume a 20% overhead for packet framing, this jumps to ~4MB/sec or ~32Mbit/sec (all network speeds are reported in megabits/sec and not megabytes/sec – a common source of confusion). Considering that the Ethernet 802.3 protocol quickly becomes saturated as the number of systems on the network increase, it might make sense to determine if the network is saturated. Of course, network switches help isolate different systems which dramatically reduces collisions allowing a network to achieve closer to its rated speed. While determining true network saturation is best done with sophisticated network analysis tools, a quick rough estimate can be obtained by using netstat and comparing the collision or error rates. If the collision or error rate is above 5-10%, consider multiple network interfaces and network bonding.

Network bonding helps alleviate this by bringing RAID-type scaling across multiple physical network cards in a technique sometimes called "RAIN" (Redundant Array of Independent Networks) to provide link aggregation. In order to work, both the host operating system and the network switch have to support the IEEE 802.3ad EtherChannel link aggregation feature via the channel bonding (OS) and port trunking (switch) features. However, as both Linux and popular CISCO switches provide these features, EtherChannel should be common on most enterprise systems. Using this technology, multiple switch ports on the same switch can be combined with multiple network interfaces on the host machine to form a single network channel that is capable of aggregating network throughput multiple times faster than a single network interface. The optimal configuration for EtherChannel is to use 2, 4 or 8 ports in a striped configuration – ideally with 4 or 8 ports the best.

OS Tuning Considerations

Note that in implementing EtherChannel, the OS may need to have some significant tuning for TCP related parameters to ensure that the OS stack does not limit the aggregated channel due to low memory used for TCP processing. For example, consider the following Linux kernel settings:

```
#Range of local ports for outgoing connections. Quite small by default: 1024 4999
net.ipv4.ip_local_port_range = 16384 65536

# tune network tcp read/write memory buffers for send/recv packets
# descriptions from ip-sysctl.txt (Linux source)

# This sets the default OS receive buffer size for all types of connections/protocols
# such as both UDP and TCP and non-TCP protocols.
net.core.rmem_default = 262144

# This sets the max OS receive buffer size for all types of connections/protocols
# such as both UDP and TCP and non-TCP protocols.
net.core.rmem_max = 16777216

# This sets the default OS send buffer size for all types of connections/protocols
# such as both UDP and TCP and non-TCP protocols.
net.core.wmem_default = 262144

# This sets the max OS send buffer size for all types of connections/protocols
# such as both UDP and TCP and non-TCP protocols.
net.core.wmem_max = 16777216


#tcp_mem - vector of 3 INTEGERs: min, pressure, max
#        low: below this number of pages TCP is not bothered about its
#        memory appetite.
#
#        pressure: when amount of memory allocated by TCP exceeds this number
#        of pages, TCP moderates its memory consumption and enters memory
#        pressure mode, which is exited when memory consumption falls
```

```
#        under "low".
#
#        high: number of pages allowed for queueing by all TCP sockets.
#
#        Defaults are calculated at boot time from amount of available
#        memory.
#
#  Notes:  some writings on web say the defaults are fine, but I have seen
# on 32GB systems where the values are (in 4K pages)
#
# net.ipv4.tcp_mem = 196608       262144   393216
#
# …the "high" value being pretty high (>1GB) when viewed in terms of 4K pages.
# On larger enterprise systems with >64GB of memory, you might want to
# constrain this to 2GB of memory.  Remember, this is in 4K pages – whereas
# the next ones (tcp_wmem and tcp_rmem) are in bytes
net.ipv4.tcp_mem = 262144 262144 524288

#tcp_wmem - vector of 3 INTEGERs: min, default, max
#        min: Amount of memory reserved for send buffers for TCP socket.
#        Each TCP socket has rights to use it due to fact of its birth.
#        Default: 4K
#
#        default: Amount of memory allowed for send buffers for TCP socket
#        by default. This value overrides net.core.wmem_default used
#        by other protocols, it is usually lower than net.core.wmem_default.
#        Default: 16K
#
#        max: Maximal amount of memory allowed for automatically selected
#        send buffers for TCP socket. This value does not override
#        net.core.wmem_max, "static" selection via SO_SNDBUF does not use this.
#        Default: 128K
net.ipv4.tcp_wmem = 262144 262144 16777216

#tcp_rmem - vector of 3 INTEGERs: min, default, max
#        min: Minimal size of receive buffer used by TCP sockets.
#        It is guaranteed to each TCP socket, even under moderate memory
#        pressure.
#        Default: 8K
#
#        default: default size of receive buffer used by TCP sockets.
#        This value overrides net.core.rmem_default used by other protocols.
#        Default: 87380 bytes. This value results in window of 65535 with
#        default setting of tcp_adv_win_scale and tcp_app_win:0 and a bit
#        less for default tcp_app_win. See below about these variables.
#
#        max: maximal size of receive buffer allowed for automatically
#        selected receiver buffers for TCP socket. This value does not override
#        net.core.rmem_max, "static" selection via SO_RCVBUF does not use this.
#        Default: 87380*2 bytes..
net.ipv4.tcp_rmem = 4194304 4194304 16777216


# Maximal number of remembered connection requests, which are
# still did not receive an acknowledgment from connecting client.
# Default value is 1024 for systems with more than 128Mb of memory,
# and 128 for low memory machines. If server suffers of overload,
# try to increase this number
net.ipv4.tcp_max_syn_backlog = 1280
# help eliminate idle connections
net.ipv4.tcp_syncookies = 1


# don't cache ssthresh from previous connection
net.ipv4.tcp_no_metrics_save = 1
net.ipv4.tcp_moderate_rcvbuf = 1
# recommended to increase this for 1 GigE …for 10 GigE use 30000
net.core.netdev_max_backlog = 2500


# How frequent probes are retransmitted, when a probe isn't acknowledged. Default: 75 seconds.
net.ipv4.tcp_keepalive_intvl = 75

# How many keepalive probes TCP will send, until it decides that the connection is broken.
# Default value: 9. Multiplied with tcp_keepalive_intvl, this gives the time a link can be
# non-responsive after a keepalive has been sent.
net.ipv4.tcp_keepalive_probes = 9

# How often TCP sends out keepalive messages when keepalive is enabled. Default: 7200 secs
net.ipv4.tcp_keepalive_time = 1800

# Ignore broadcast messages
net.ipv4.icmp_echo_ignore_broadcasts = 1
```

Recommendations for ASE

While ASE can support multiple listeners and provide a means of scaling across multiple adapters, the reality is that most ASE systems are implemented with only a single network listener for public (non-administrator) access. This often means a single network interface is handling all the traffic for end-users. However, most systems are composed of multiple different applications or classes of users (e.g. internal vs. internet). In such cases, it would make sense to have multiple listeners configured in ASE simply to reduce the network latency to the various applications or types of users. This latter is often caused by application consolidation onto the newer, faster hardware. However, the previous independent systems often supported at least two dedicated network interfaces – while often the new host machine supports most likely only two or four network interfaces by default.

EtherChannel or network bonding is strongly encouraged for ASE public networks as well as private networks for ASE Cluster Edition. Quad port 1GbE cards such as the Intel PRO/1000 PT Quad are less than $500 US at the time of writing. Interestingly, this is about the same price as single port 10 Gigabit Ethernet card such as the Intel E10G41AT2 with dual port cards listing for about $1,200 US. A 10GbE switch is <$5000 US – so in some cases it might be far more cost effective to simply upgrade to 10GbE instead of using channel bonding. In cases where an upgrade is impossible or impractical, using one or two quad port 1GbE cards with channel bonding may provide a cost effective solution to network limitations.

OS Services & DBMS Functionality

Unfortunately, most DBA's do not pay attention to the environment their DBMS runs in. The proper tuning of the OS environment can improve throughput by a considerable percentage – 20% or greater. Similarly, running the DBMS in a virtualized environment can possibly negatively impact performance.

DBMS Architecture & OS Environment

One of the most overlooked aspects is the OS environment – specifically any kernel tuning or hardware configuration tuning.

OS Kernel Tuning

Historically, Sybase has taken the position during benchmarks to run its tests on out of the box OS configurations with the assumption that the enterprise OS's were reasonably tuned and that by not requiring any specific tuning, ASE would not depend on kernel tuning that might conflict with other applications sharing the hardware. It also sometimes assumed that other applications would not also adjust kernel configurations to interfere with ASE. However, this often means that customer systems were unoptimized – sometimes significantly.

Enterprise Unix

For example, Sun often used the following configurations for SPC-1 benchmarks – a benchmark by the Storage Performance Council (www.storageperformance.org) – which included an OLTP workload simulation:

```
-- snippet from /etc/system for a SUN SunFire 4800 SPC-1 FDC dated April 7, 2003

* Memory allocation parameters
set vxio:voliomem_chunk_size = 1048576
set vxio:voliomem_maxpool_sz = 134217728
* I/O related parameters
set vxio:vol_default_iodelay = 10
set vxio:vol_maxkiocount = 32768
set vxio:vol_maxioctl = 131072
set vxio:vol_maxio = 8192
set vxio:vol_maxspecialio = 10240
* VM related
* scanner I/Os per second page-outs.(default 65536 for E10000)
* set maxpgio = 16384
* # of pages the scanned when freelist falls below lotsfree.
* set to 1/16 to 1/4 RAM up to 1Gb/sec (131072)
*set fastscan = 65536
**Fri Mar 31 16:07:32 PST 2000
* R. McDougall: increase maxpgio to prevent the scanner from limiting writes
set maxpgio = 65536
* R. McDougall: increase fastscan to limit the effect the page scanner
* has on file system throughput
set fastscan = 65536
* increase capability to do 1Mb IOs to *raw* devices, 32MB max.
*set maxphys = 1048576
set maxphys = 4194304
* TCP related
* decrease potential connection backlog by:
* 1. increase connection hash table size, increase if connections are high.
set tcp:tcp_conn_hash_size = 32768
* 2. depth of destination queue (# of messages outbound streams can hold)
* NOTE: sq_max_size = 0 is unlimited, but uses more kernel memory.
set sq_max_size = 10
* vxvm_START (do not remove)
forceload: drv/vxdmp
forceload: drv/vxio
forceload: drv/vxspec
* vxvm_END (do not remove)
```

The use of such an old report is somewhat deliberate as it shows that this information was publically available for years – information that can improve the scalability of even mid-range systems as customers using these settings on systems reported 20% performance gains. Such OS configurations control:

- How quickly the OS will poll for completed IO's. For example, the default setting for set vxio:vol_default_iodelay is 50 – which is based on the assumption of internal SCSI hard drives. SAN's and caching controllers are much quicker – which can improve the IO response times – but only if the OS kernel limit is reduced. Example above shows this reduced to 10 – a possible 5x improvement in IO response times.
- How many concurrent IO's are allowed per volume. For example, the default setting for vxio:vol_maxkiocount restricts the OS to only submitting 2,048 concurrent IO's. If that volume happens to be the one with the raidset(s) supporting the high transaction tables, the OS may be limiting the DBMS's scalability. Example above shows this set to 32768 – a 16x increase in concurrent IOs allowed.
- The network receive tuning for TCP. The above illustrates increasing the hash table size for supporting high numbers of connections from the default of 512 to 32K. In addition to increasing the hash table to support the number of connections, you may also need to increase the receive memory using tcp_recv_hiwat (and tcp_max_buf) from the default of 24KB (and 8KB) to something more realistic.
- The network send tuning for TCP. The above illustrates changing the queue depth – however, like with network receive, there may be other network send parameters such as tcp_xmit_hiwat which has a default of 16KB.

Every OS has different tuning parameters – and these can change between OS releases (above are mostly Solaris 8 & 9). Other OS's such as HP-UX suggest running utilities such as chatr that change the binary slightly for memory access reasons. Some of these OS specific configuration specifics are listed in the following papers available from Sybase's web site:

Optimizing ASE 15 on HP-UX Integrity Server rx7640 – While it references the rx7640 specifically, the tuning suggestions are applicable to all HP-IA (Itanium 2) based platforms. Reported a 30% gain over the default configurations.

Optimizing Sybase® ASE for IBM AIX: An IBM and Sybase white paper on optimizing Sybase ASE for AIX 5L and AIX 6.1 – (Revised August 2009). Discusses AIX kernel parameters, file system considerations and general ASE tuning as relates to IBM AIX.

Note that more and more hardware vendors (less Linux) are starting to realize that the primary deployment for their hardware platforms is as DBMS machines. As a result, some are starting to tune the kernel such that DBMS users will not need to make any kernel changes to get optimal performance (such as IBM with AIX 6L).

Linux

Linux is a bit different. First, the OS is primarily targeted at development and web/application server deployment environments with discrete users accessing individual files. While there is kernel settings for network, very few IO tuning parameters exist beyond fs.aio-max-nr. In addition to the network kernel settings above, the following kernel tuning settings should be considered for ASE:

```
# must be higher than ('disk i/o structures'+ 'max asyncio per engine')* 'max online engines'
# or best just to set to max conceivable (1MB)
fs.aio-max-nr = 1048576

#each device plus each concurrent user will need a file handle - plus those used by the
#software…plus factor in file system devices…default is 3MB
#or best to just set to 2x higher than default
fs.file-max = 6291456

#standard shared memory tuning…
#first disable shared memory security blocks
kernel.exec-shield = 0
kernel.randomize_va_space = 0

# shmall is the total amount of shared memory allowed (in 4k pages) - period
# attempts to allocate shared memory above this will fail.  Multiply by 4096
# to see bytes
kernel.shmall = 4294967296

# shmmax is the largest shared mem segment size (in bytes) (64GB)  ASE, Backup
# server and other Sybase processes can use multiple shared memory segments -
# each segment can be up to shmmax in size (but may be smaller due to free
# contiguous memory space).  Unless a small box, set in even GB.
kernel.shmmax = 68719476736

#maximum number of shared memory segments system wide
kernel.shmmni = 4096

# ASE 15.0.3 and higher can use huge pages - do NOT use for ASE's earlier than
# 15.0.3 as this will block the memory from being available to ASE.  Each huge
# page equivalates to 2MB of memory - but ASE will round shmmax up to nearest 256MB
# as a result (so make sure shmall is higher than result).
vm.nr_hugepages = 2048
```

These should be entered in /etc/sysctl.conf and then can be loaded using the sysctl –p command as normal.

However, the on-again/off-again support of raw devices in Linux (originally to be deprecated in later RHEL 4.x and early 5.x kernels, but as of RHEL 5.4, raw partition support is now back in the upstream kernel) and the multiple flavors of file systems requires a lot more tuning configurations around the file system journaling, block size, inode ratios, etc. However, there are some similarities with other OS's – but the OS limits are initially much, much lower as would be expected from an OS that has its roots in a desktop/small server OS. For example, in Solaris, the number of concurrent I/O's per logical volume is controlled by vxio:vol_maxkiocount – which defaults to 2048 – a number that most Solaris admins admit is very low and workstation level. AIX has a similar concept with queue depth or queue length – which on page 17 of the above IBM whitepaper suggests starting at 8192 and suggesting higher may be necessary. In Linux, the same parameter is controlled at a device level with /sys/block/<devname>/queue/nr_requests – which defaults to 128. This is substantially lower than similar enterprise Unix systems that Linux is often replacing. The behavior is quite different as well. With the other OS's, if ASE attempted to make another I/O call and the request queue was full, it would simply be denied. In Linux, ASE blocks in io_submit(). In other words, the call stops being "asynchronous" and would instead wait until the number of outstanding I/O's to the given block device dropped below the threshold – at which point ASE immediately submits the pending IO and the problem continues. This latter aspect may be due

to the fact it is a block device whereas on other enterprise Unix systems, character (raw) devices are more common. As a result, on Linux systems, it is strongly recommended that you include a script in rc.init to do the following:

```
-- do for each filesystem device used by Sybase for either data or backups
echo 1024 > /sys/block/<devname>/queue/nr_requests
```

In addition, Linux supports the notion of different I/O schedulers (aka disk elevators) that control performance characteristics (particularly those controlled by the OS kernel in other OS's) as well as the write ordering – a possible scary thought when working with DBMS's striving to maintain ACID. This I/O reordering is an attempt to reorder I/O's submitted to a particular device in order to minimize head movement between tracks. This is a kernel tuning feature that is likely irrelevant and faulty when using RAID or SAN devices (as well as SSD's) – and may not be all that useful in modern day disks with Native Command Queuing (NCQ) (which reorders the I/O's by disk physical location within the disk drive interface itself). The current Linux kernel 2.6 I/O schedulers for block (filesystem or block device I/O) are:

Deadline Scheduler (deadline) – Aggressively reorders I/O to using 5 I/O queues while assigning a per-request expiration deadline to avoid I/O starvation. Reads are given a higher preference than write operations in the I/O reordering. I/O's are sorted by logical block or by the deadline expiration using 4 of the queues while the 5th queue is used to dispatch to the device driver. In reality, only reads are assigned deadlines, so the goal is to attempt to satisfy reads within a time frame, while writes have no deadline expiration and could be delayed extensively.

Anticipatory Scheduler (anticipatory) – (old default) Based on the deadline scheduler above, attempts to reduce the per thread/process I/O response time by delaying further I/O requests to the same device to see if the I/O requester that just completed a read operation will submit another read that is "close" (block-wise) to the first one. The definition of "close" is relative to the delay – as the delay time increases, the distance increases proportionately. If so, this request is handled ahead of any other request. Only when "close" read operations have dried up are write operations processed. The obvious problem with this scheduler is that it assumes the blocks being read by the process are contiguous blocks such as initiated by a single user reading a single file vs. a server process acting on the behalf of multiple users performing random I/O operations.

Completely Fair Scheduler (cfq) – (default as of kernel 2.6.18) This scheduler attempts fair allocation of all the I/O bandwidth among all the I/O requestors. This scheme uses a hashing method on the process id (pid) to place the I/O request into one of 64 I/O queues. I/O's are moved from one of these queues to a dispatch queue in round-robin fashion. While a block of I/O's are being moved to the dispatch queues, they are sorted by logical block to minimize head seek times.

Noop Scheduler (noop) – This scheduler essentially just does basic request merging and does not attempt to re-order the I/O's. This scheduler is most appropriate for SAN-hosted, RAID or non-disk based devices (such as SSD's).

The point is that very few Linux system administrators realize that the OS out of the box is tuned completely wrong as a DBMS server when used with high-end storage systems or even low-end caching RAID controllers. An example of the impact of the IO scheduler and queue wait times can be seen in the following output from a iostat execution:

```
-- output from iostat -x
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           1.36    0.00    3.50   11.15    0.00   83.99

Device:         rrqm/s   wrqm/s   r/s   w/s   rsec/s   wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sda               0.00     8.82  0.00  1.00     0.00    78.56    78.40     0.02   24.80  24.80   2.48
sda1              0.00     0.00  0.00  0.00     0.00     0.00     0.00     0.00    0.00   0.00   0.00
sda2              0.00     8.82  0.00  1.00     0.00    78.56    78.40     0.02   24.80  24.80   2.48
sdb               0.00     0.80 41.28 12.02   330.26    98.60     8.05     1.17   21.95  17.79  94.83
sdb1              0.00     0.80 41.28 12.02   330.26    98.60     8.05     1.17   21.95  17.79  94.83
sdc               0.00     0.80  0.80 29.06     6.41   238.88     8.21     3.03  101.61   6.47  19.32
sdc1              0.00     0.80  0.80 29.06     6.41   238.88     8.21     3.03  101.61   6.47  19.32
sdd               0.00     0.00  0.00  0.00     0.00     0.00     0.00     0.00    0.00   0.00   0.00
sdd1              0.00     0.00  0.00  0.00     0.00     0.00     0.00     0.00    0.00   0.00   0.00
dm-0              0.00     0.00  0.00  9.82     0.00    78.56     8.00     0.23   23.84   2.53   2.48
dm-1              0.00     0.00  0.00  0.00     0.00     0.00     0.00     0.00    0.00   0.00   0.00
```

In addition to the extremely slow IO service times (~18ms – more than double the 8ms at which point IOs are considered slow), the IO scheduler queuing effect added insult to injury by delaying the IOs another 4ms for a total of ~22ms per IO. Note that while other devices appear to have worse times, there is little IO activity on those devices, consequently the numbers are skewed. Most of the problem in the above is likely caused by either the kernel tuning parameters around fs.aio-max-nr or the /sys/block/<devname>/queue/nr_requests issues. However, a contributing factor is the IO scheduler as evident by the '*await*' times.

While this can be changed by modifying the /boot/grub/grub.conf – it also can be changed on a per device basis by:

```
## syntax for reading the current scheduler:
## cat /sys/block/{DEVICE-NAME}/queue/scheduler

cat /sys/block/sda/queue/scheduler


## syntax for changing the current scheduler:
## echo {SCHEDULER-NAME} > /sys/block/{DEVICE-NAME}/queue/scheduler

echo noop > /sys/block/hda/queue/scheduler
```

The best way to change it is to modify /boot/grub/grub.conf and add elevator=noop as a kernel parameter:

```
# grub.conf generated by anaconda
#
# Note that you do not have to rerun grub after making changes to this file
# NOTICE:  You have a /boot partition.  This means that
#          all kernel and initrd paths are relative to /boot/, eg.
#          root (hd0,0)
#          kernel /vmlinuz-version ro root=/dev/VolGroup00/LogVol00
#          initrd /initrd-version.img
#boot=/dev/sdk
default=0
timeout=10
splashimage=(hd0,0)/grub/splash.xpm.gz
hiddenmenu
title CentOS (2.6.18-164.11.1.el5xen)
        root (hd0,0)
        kernel /xen.gz-2.6.18-164.11.1.el5
        module /vmlinuz-2.6.18-164.11.1.el5xen ro root=/dev/VolGroup00/LogVol00 rhgb elevator=noop
        module /initrd-2.6.18-164.11.1.el5xen.img
title CentOS (2.6.18-164.11.1.el5)
        root (hd0,0)
        kernel /vmlinuz-2.6.18-164.11.1.el5 ro root=/dev/VolGroup00/LogVol00 rhgb elevator=noop
        initrd /initrd-2.6.18-164.11.1.el5.img
title CentOS (2.6.18-164.el5xen)
        root (hd0,0)
        kernel /xen.gz-2.6.18-164.el5
        module /vmlinuz-2.6.18-164.el5xen ro root=/dev/VolGroup00/LogVol00 rhgb quiet
        module /initrd-2.6.18-164.el5xen.img
```

```
title CentOS-base (2.6.18-164.el5)
        root (hd0,0)
        kernel /vmlinuz-2.6.18-164.el5 ro root=/dev/VolGroup00/LogVol00 rhgb quiet
        initrd /initrd-2.6.18-164.el5.img
```

The above changes the default I/O scheduler to the noop scheduler. In addition, there are a number of I/O scheduler tuning parameters for each I/O scheduler that may need to be tuned to provide optimal I/O.

Hardware Configuration

In addition to the OS kernel tuning, there is the often hidden joy of tuning the hardware itself. For example, IBM HBA cards to connect IBM hosts to SANs are shipped with a default configuration that only supports 255 concurrent I/O's. Additionally, the tuning of the SAN cache itself can be a challenge and requires understanding the type of database application and IO patterns. While the default might be tempting split the cache 50:50 between 'Read' and 'Write' cache, as mentioned earlier, DBMS's with OLTP workloads experience a random I/O pattern that pretty much negates the usefulness of 'Read' cache. Additionally, most SAN systems have cache utilization thresholds, such as when 33% of the Write Cache is utilized, the SAN starts de-prioritizing host read requests and at 60%, it may stop processing read requests altogether. The smaller the SAN write cache, the more quickly this threshold is reached.

Recommendations

DBA's wishing to optimize their hardware environment may want to do the following:

- Download any Storage Performance Council (SPC) benchmark Full Disclosure Reports (FDR) for their OS – particularly if their storage subsystem was the one under stress. (www.storageperformance.org)
- Download any Transaction Performance Council (TPC) benchmark Full Disclosure Reports (FDR) for their OS – particularly if their platform was the one under stress (DBMS and app server combination are irrelevant – focus OS version and platform). OS tuning parameters are typically near the end of Appendix C. (www.tpc.org)
- Download a copy of their OS's kernel tuning and file system tuning manuals (if using file systems with DIRECTIO) and review with system administrators the tuning parameters related to disk and network IO, process control (such as threads per process), etc.
- Discuss the OS tuning with both the host vendor pre-sales support staff as well as the storage vendor pre-sales support staff.

…and of course test the configurations with a representative multi-user workload.

Virtualization & Consolidation

One of the biggest trends in recent years has been the trend of "virtualization" – which unfortunately seems have adopted a singular reference to multiple guest virtual machines running in separate "partitions", "domains", "VM's", "LPAR's", etc. on the same host. In theory, the hardware was grossly underutilized, and that by consolidating applications from multiple

hosts onto a single faster or larger host, significant savings in datacenter space, power and cooling can be achieved. The supposition is that the impact of running in a virtualized environment is minimal on the applications being consolidated. The reality can be quite a bit different – depending on the type of machine partitioning used.

This concept actually is important. A simpler methodology would be to simply install all the applications in the single host without using virtualization. However, experience with this in the past has shown that some applications will monopolize the resources (memory, CPU, IO/network bandwidth) which very negatively impacts the other applications. Partitioning off portions of capacity and constrain demanding applications restores the performance for all the applications.

The reason this is important is that a DBMS is similar in a lot of ways to an operating system. Unfortunately, on many large SMP environments, DBA's simply allow all the applications to co-exist without any attempt to constrain applications – and the results have been the same as with large SMP hosts and multiple applications – chaos and performance issues.

Hard Partitions

A "hard" partition is the most transparent to the applications as it provides dedicated system resources and often a separate OS image. As a result, hard partitions may be limited to division based on the number of system boards in the host as well as whether or not each system board has a network, disk I/O and amount of memory available to it. While a hard partition can be resized, it often requires a reboot of the entire host machine. The exact restrictions on hardware configuration and resizing are often OS dependent.

The reason a "hard" partition is the most transparent from an application performance perspective is that there is a single OS layer between the application and hardware and the hardware is dedicated – not shared. The result is the same as if the application was installed on a dedicated host machine of the same size as the partition.

One advantage of using hard partitions is that some vendors, such as IBM, allow different partitions on the same host to tunnel their network communications through the system backplane. As a result, applications that frequently have network interactions benefit from an enormous gain in network speed and bandwidth – as well as a separation of client network communications from server to server network communications.

Containers & Soft Partitions

Hard partitioning larger systems was often the first partitioning option that hardware vendors supported. However, because of the inflexibility and lack of dynamic resizing, most began offering some form of "soft" partitioning or "containers". The exact features and capabilities of a "soft" partition is also OS vendor dependent, however, most offer the ability to:

- Constrain the amount of memory and CPU available to a single partition
- Dynamically resize the partition without a reboot
- Constrain applications to only run within associated partitions

One often difference between soft and hard partitions is that while each hard partition has its own OS image, in soft partitioning, there is a single OS image on the host machine. The partitioning is implemented as OS services within the operating system. As a result, soft partitioning is extremely flexible if the performance requirements change frequently. For example, some applications might have a low usage during the day, but need a lot more resources at night for batch processing. Meanwhile, other applications have the opposite profile.

Some OS's also support the ability to constrain applications without creating partitions by supporting features that restrict a single resource such as CPU. For example, Sun Solaris supports both "zones" and "containers" as a means of soft partitioning as well as "processor sets". A processor set is simply the ability to constrain a particular application and any spawned threads/processes to a set of virtual processors. Note that this does not constrain memory. Sometimes this is used to complement partitions in order to better optimize the virtual processors in such a way that L2 system cache flushes are avoided. Another common use is to constrain one application from another when both are in the same region.

Virtual Machines

One of the most frequently recognized forms of OS services for virtualization today is virtual machines. Initially made popular by VMWare™, other OS's now include native support such as HPUX and Linux. We are all familiar with the concept in which an OS hypervisor supports running a self contained virtual machine with a guest operating system as well as installed applications. The hypervisor can run either on bare hardware (native VM) or it can run as an application within the host OS (hosted VM – the most common). Storage can be within the virtual machine or mounted when it runs.

One of the keys to virtual machines is the virtual machine server. Generally, a system that has been partitioned has at least 2 or more virtual machines. In order to optimally configure the workload to ensure that the individual applications within the virtual machine get the correct amount of resource time, a virtual machine server needs to provide some form of workload management.

Another key to virtual machines is the type of virtualization with respect to OS calls. At its basic level, a hosted VM hypervisor needs to be able to translate the guest OS calls into the OS calls used by the host OS. This can add considerable overhead for application performance. As a result, anecdotal evidence suggests that a VM hosted DBMS will suffer at least a 20% degradation over the same DBMS running on a host with the same resources as a VM.

However, virtual machines have advantages over OS partitions in that they can be transportable. As a result, virtual machines can be moved quite quickly from one host machine to another with very little downtime. A key thing to remember is that there still is downtime. For a DBMS of any considerable size, this could result in performance degradation for hours as the system ramps back up the data cache, etc.

DBMS Functionality

Earlier, it was stated that the DBMS system is similar to the OS in its need for providing partitioning or process segmentation. One of the biggest hurdles is that while system administrators have long recognized the need for OS partitions on large machines with multiple applications, they generally have been thinking of isolated and completely contained applications. The next leap is to consider that most applications these days are some flavor of client-server – which means that a good portion of the application's logic and workload happens on the server. When multiple disparate applications share the same DBMS server, it is similar to the effect of multiple applications sharing the same host. The more applications and the larger the DBMS environment, the more critical this becomes.

Another hurdle is recognizing what is an application vs. a workload. Often times, DBA's and system administrators will consider a fairly broad definition for application – without breaking down the application modules or other aspects that depict different workloads. In fact, often, multiple different executables are often considered a single application by the DBA's – e.g. the java middle-tier components, batch scripts, business objects reports, web-services, etc. The net result often is a system that doesn't scale well simply due to unnecessary contention.

A scalable DBMS platform can help alleviate this problem by providing:

Scalable & Tunable I/O Subsystem – in particular one that maximizes the ability to do asynchronous/non-blocking writes and prevents cache stalls

Tunable Concurrency Controls – the ability to increase the number of concurrency controls to avoid contention for serialization elements while reducing the overhead and risks of too many concurrency controls.

Optimized and Relaxed Durability – the ability to use either fully relaxed or slightly relaxed durability controls to reduce the contention on the DBMS transaction logs.

Logical Memory Management – the ability to separate critical, normal and non-critical data to minimize physical reads for critical data, support large I/O's and support multiple users trying to modify the buffer chain simultaneously.

Logical Process Management – the ability to prioritize different user tasks according to their relative importance as well as isolate critical processes from normal processing or restrict non-mission critical processes in order to avoid problems with mission critical tasks.

Workspace Segmentation – the ability to separate the scratch space for intermediate processing results from other applications – providing scalability through I/O capacity for temporary workspace as well as improving availability by avoiding the risk of a single task filling all the available temporary workspace.

Workload Management – the ability to manage workloads by partitioning the resources as well as runtime dynamic rebalancing the workload.

Sybase ASE provides all of these features. The following sections describe the details of how they work as well as configuration and tuning suggestions

Tuning System Tasks & Spinlocks

Tuning the DBMS is always somewhat of a black art. The reason is that the optimal tuning configuration often depends on a number of factors, including the application characteristics, the mix of applications, the number of concurrent queries, the amount of memory available, etc.

If tuning the DBMS has been a black art, tuning the DBMS internals often remains a mystery to DBA's and often is an area left unturned or just guessed at from previous installations, "web gouge" (not always accurate) or on someone else's say-so (equally unreliable). Many of these internal settings have a direct impact on scalability – and more to the point can be often derived more easily than some of the more application oriented settings DBA's are more familiar with.

I/O Subsystem Tuning

The I/O subsystem within ASE consists of a number of different features and system tasks:

- I/O structures such as disk i/o structures, max async i/o per server, max async i/o per engine
- Housekeeper & Checkpoint processes
- Asynchronous prefetch utilities

The first one is a set of configuration parameters that DBA's are probably the most familiar with of all the internal settings. The '*disk i/o structures*' configuration parameter has a direct correlation with the number concurrent asynchronous I/O's that ASE can submit at one time. The '*max async i/os per server*' and '*max async i/os per engine'* settings also do as well, but depends on the OS which ones apply.

Disk I/O Structures

For each I/O requested – whether asynchronous or synchronous – ASE creates a disk I/O structure – essentially, a data structure containing a pointer to the pages in memory, the virtual device and the logical blocks (or offset within the virtual device) requested. Note that the disk I/O structure does not contain the data itself as ASE will use DMA access to write the data to the device. The disk I/O structure is filled out when the I/O is first requested (queued) and is only released when the I/O returns. Consequently, the maximum number of I/O's an ASE server can submit concurrently or have outstanding at any point in time is limited by the number of disk I/O structures. To avoid contention on allocating disk I/O structures during runtime in SMP environments, half of the allocated disk I/O structures are distributed evenly at start-up between the engines to create an per-engine local pool of disk I/O structures. The other half are retained in the disk I/O structure global pool.

```
local engine pool = ((‘disk i/o structures’/2)) / ‘max online engines’

-- assume a 16 engine server and a disk i/o structure setting of 1024
local engine pool = ((1024/2))/16
local engine pool = 512/16
local engine pool = 32
```

If an engine exhausts it's allocated local engine disk I/O structure pool, it can request more from the global pool. However, because this is an expensive operation, often the task will also migrate to a different engine. The number of IO requests that have been queued over time can be monitored via monIOQueue:

```
create existing table monIOQueue (
        InstanceID                      tinyint,
        IOs                             int,
        IOTime                          int,
        LogicalName                     varchar(30) NULL,
        IOType                          varchar(12) NULL,
)
external procedure
at "@SERVER@...$monIOQueue"
go
grant select on monIOQueue to mon_role
go
```

However, note that like many MDA values, monIOQueue tracks cumulative I/O requests. This can be determined by checking the Indicators bit in monTableColumns to see if the "wrapping" bit is set:

```
1> select ColumnID, ColumnName, Indicators, WrappingBit=Indicators&1, Description
2>   from master..monTableColumns
1>  where TableName='monIOQueue'
2>  go
 ColumnID ColumnName    Indicators  WrappingBit Description
 -------- ------------- ----------- ----------- -----------------------------------------------------
        0 InstanceID              0           0 The Server Instance Identifier (cluster only)
        1 IOs                     1           1 Total number of I/O operations
        2 IOTime                  1           1 Total amount of time (in milliseconds) spent waiting for I/O
        3 LogicalName             0           0 Logical name of the device
        4 IOType                  0           0 Category for grouping I/O ['UserData', 'UserLog', 'TempdbData',
'TempdbLog']

(5 rows affected)
```

Finding out if the disk I/O structures configuration is too low can be a bit challenging as a result. One way is that sp_sysmon number of outstanding I/O's per engine during its execution:

```
Disk I/O Management
-------------------

  Max Outstanding I/Os            per sec      per xact       count  % of total
  --------------------------  ------------  ------------  ----------  ----------
    Server                             n/a           n/a         326         n/a
    Engine 0                           n/a           n/a         240         n/a
    Engine 1                           n/a           n/a         154         n/a
    Engine 2                           n/a           n/a         229         n/a
    Engine 3                           n/a           n/a         206         n/a
    Engine 4                           n/a           n/a         168         n/a
    Engine 5                           n/a           n/a         144         n/a
    Engine 6                           n/a           n/a         152         n/a
    Engine 7                           n/a           n/a         161         n/a
    Engine 8                           n/a           n/a         161         n/a
    Engine 9                           n/a           n/a         153         n/a
    Engine 10                          n/a           n/a         154         n/a
    Engine 11                          n/a           n/a         152         n/a
    Engine 12                          n/a           n/a         155         n/a
    Engine 13                          n/a           n/a         129         n/a
    Engine 14                          n/a           n/a         147         n/a
    Engine 15                          n/a           n/a         147         n/a
    Engine 16                          n/a           n/a         130         n/a
    Engine 17                          n/a           n/a         140         n/a
    Engine 18                          n/a           n/a         177         n/a
    Engine 19                          n/a           n/a         144         n/a
    Engine 20                          n/a           n/a         132         n/a
    Engine 21                          n/a           n/a         148         n/a
    Engine 22                          n/a           n/a         185         n/a
    Engine 23                          n/a           n/a         188         n/a
    Engine 24                          n/a           n/a         133         n/a
    Engine 25                          n/a           n/a         127         n/a
    Engine 26                          n/a           n/a         231         n/a
    Engine 27                          n/a           n/a         234         n/a
    Engine 28                          n/a           n/a         249         n/a
    Engine 29                          n/a           n/a         161         n/a
    Engine 30                          n/a           n/a         160         n/a
    Engine 31                          n/a           n/a         233         n/a
    Engine 32                          n/a           n/a         243         n/a
```

```
  Engine 33                          n/a          n/a          139       n/a
  Engine 34                          n/a          n/a          245       n/a
  Engine 35                          n/a          n/a          237       n/a
  Engine 36                          n/a          n/a          235       n/a
  Engine 37                          n/a          n/a          224       n/a
  Engine 38                          n/a          n/a          235       n/a
  Engine 39                          n/a          n/a          139       n/a
  Engine 40                          n/a          n/a          247       n/a
  Engine 41                          n/a          n/a          147       n/a
  Engine 42                          n/a          n/a          146       n/a
  Engine 43                          n/a          n/a          145       n/a
  Engine 44                          n/a          n/a          254       n/a
  Engine 45                          n/a          n/a          165       n/a
  Engine 46                          n/a          n/a          106       n/a
  Engine 47                          n/a          n/a          131       n/a
  Engine 48                          n/a          n/a          176       n/a
  Engine 49                          n/a          n/a          177       n/a
  Engine 50                          n/a          n/a          210       n/a
  Engine 51                          n/a          n/a          187       n/a


I/Os Delayed by
  Disk I/O Structures                n/a          n/a            0       n/a
  Server Config Limit                n/a          n/a            0       n/a
  Engine Config Limit                n/a          n/a            0       n/a
  Operating System Limit             n/a          n/a            0       n/a


Total Requested Disk I/Os          4159.9          1.5      7541885
```

Note that the first line reports the maximum outstanding I/O's across the entire server during the sp_sysmon duration while the others report the maximum outstanding for each engine. By just looking at the server line, it would appear as if only 326 disk I/O structures would be all that would be needed as that is the maximum outstanding. However, note that each engine at some point had between 106 and 254 outstanding I/O's. Given the engine local pool, ideally, we would need at least 106 disk i/o structures per engine and preferably ~256 disk i/o structures per engine in the local engine pool to avoid having to incur the expense of grabbing from the global pool. With a total of 52 engines, we would need (52*256)*2 or 26,624 disk I/O structures in the configuration file. Note that the multiplication by 2 was to offset the fact that only half of total disk I/O structures are allocated to the local engine pool – consequently, if we have a total of 512 per engine, 256 per engine would be allocated to the global pool. A good starting point for configuration is 300-400 disk I/O structures per engine – and after monitoring, this may need to be increased.

Max Async I/Os per Engine/Server

If the disk I/O structures can be pictured as the interface between the cache and the virtual device, the asynchronous I/O settings can be thought of as the interface between the virtual device and the operating system. Every operating system preallocates an amount of memory to be used in tracking asynchronous IO operations – but it may be on the entire OS level or on a per device level. For example, as noted earlier, for Linux, this is fs.aio-max-nr, which applies to the entire OS. For Solaris, the setting is controlled on a per volume basis by the configuration vxio:vol_maxkiocount; for AIX it is the aio maxreqs (now in the ioo command set in AIX 6.1+) which is limits the concurrent I/O's per LPAR. Note that all of this applies to raw partitions. When using file systems devices – whether DIRECTIO or not – then additional limits may apply. For example, the default HP file system in HPUX is synchronous – which causes a huge performance penalty. In HP-UX 11, HP started shipping the Veritas File System as a method of providing an asynchronous as well as journaled file system. However, the Veritas File System has several tunables such as *max_diskq* and *oltp_load*.

In general, some operating systems require that processes performing asynchronous I/O operations reserve a number of AIO descriptors – analogous to file descriptors. This is what the tuning parameters 'max async i/os per engine' and 'max async i/os per server' relate to – the number of AIO descriptors that ASE will reserve at boot time. In some OS's the number is unlimited, and consequently the default for these values is ~2 billion. For OS's that require tuning this value, the number should be less than the number of total number of asynchronous I/O's allowed by the operating system – a bit of a challenge to determine if the limit is controlled by volume or file system when the server may be issuing I/Os to multiple file systems or volumes. Consider the following table:

OS/File System	Limit Enforced:	I/O Limit Location	Default Value
MS Windows NT/Server	Controller Driver per Device	HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\<*Miniport_Adapter*>\Parameters\<*DeviceNum*>\NumberOfRequests (REG_DWORD)	(set by driver installation)
Linux	OS kernel	/etc/sysctl.conf → fs.aio_max_nr	65536
Linux File Systems	Device	/sys/block/<devname>/queue/nr_requests	128
AIX Raw Devices	LPAR	SMIT → aio maxreqs	16384
HPUX Raw Devices (SCSI)	SCSI Device/LUN	scsimgr → max_q_depth	8
HPUX Raw Devices (aio driver)	Async Disks	asyncdsk → max_async_ports	50
HPUX Raw Devices (POSIX)	OS kernel	SAM → aio_max_ops	2048
Solaris Raw Devices	OS Volume	/etc/system → vxio:vol_maxkiocount	2048
Veritas File Systems (e.g. HPUX)	File System File	vxtunefs → max_diskq & oltp_load	1MB & 0 (off)

As shown above with the HP example, a lot can depend on the type of device, which driver is used, and whether a file system device or raw partition. The limits can also be implemented in the number of requests, the amount of data to be written or the number of ports used to communicate the array of requests to the device. Consequently, with each OS being different, it is best to determine the appropriate setting in conjunction with system administration staff. To do so, note that Windows and Solaris tend to use the 'max async i/os per server' configuration, while other OS's will use the 'max async i/os per engine'. The reason Windows uses the per server setting is that there is a single parent process and each engine is a thread.

As of ASE 15.0.3 ESD #1, the current calculation that should be used to determine the OS setting is:

```
-- ASE 15.0.3 ESD 1+
Max concurrent AIOs = (disk i/o structures + max async ios per engine) * number of engines

-- Prior to ASE 15.0.3 ESD 1 and future (TBD)
Max concurrent AIOs = (disk i/o structures) * number of engines
```

The computed value from the above should be used when adjusting the various OS configurations – taking into consideration whether tuning the OS kernel or a device. OS kernel settings should be at least comfortably higher than the above calculation, while device settings should be appropriate for the amount of device cache and device speed (e.g. normal disk vs. SSD). If the OS kernel cannot be adjusted high enough, you may need to use 'max async i/os per engine/server' to govern the ASE to avoid exceeding the OS limitations.

Prior to submitting the IO to the OS, the number of outstanding requests would be checked compared to the ASE configuration. The way it works is as follows:

- Concurrent access to devices is controlled via a semaphore on each Sybase virtual device. In older versions of ASE, this was used to ensure device mirroring sequencing. Mostly, today, this is used to track the

outstanding I/O's and write sequencing. Each task that wants to write to a device must grab the device semaphore if ASE disk mirroring is not disabled or if the system exhausts the number of disk structures.

- Before a dirty page can be written to disk, a disk i/o structure must be obtained from the engine initiating the I/O operation. .
- Once the disk i/o structure is filled out, the i/o is submitted to the OS for that device using an available aio structure limited by the '*max async i/os per server*' (Solaris & MS Windows) or the '*max async i/os per engine'* (all other OS's) configuration setting as well as the OS limitations on concurrent asynchronous IO operations via limits on AIO descriptors.
- I/O's are submitted via the OS aio_read() and aio_write() calls. Each loop through the scheduler as well as each '*i/o polling process count*', the engines check for completed i/o's (from any engine) using aiowait() as a polling mechanism.
- When an I/O has completed, the ASE kernel "wakes" up the spid by moving it from the sleep queue to the appropriate "runnable queue" for the priority of the process (mostly this will be the EC2/medium run queue).

Remember: asynchronous write I/O's are mostly initiated by the housekeeper and checkpoint tasks while read I/O's and synchronous writes are initiated by user tasks, although user tasks can initiate asynchronous writes due to causing a dirty buffer to cross the wash marker.

The question is if the disk i/o structures configuration setting controls the number of outstanding I/Os, what should be the configuration for 'max async i/os per engine'?? While it might be tempting to set it to (disk i/o structures)/(number of engines), the result would be that batch processes or bulk load operations would be limited to only a portion of the possible I/O's that the overall system could handle – especially if the I/O's are impacting multiple devices and could handle increased concurrency. As a result, the following rules of thumb should be used in setting the per engine and per server values:

- If the default 'max async i/os per engine' and 'max async i/os per server' are unlimited (~2 billion), do not make any changes – tune the OS/file system appropriately.
- Otherwise, set both the 'max async i/os per engine' and 'max async i/os per server' to the same value as 'disk i/o structures' and tune the OS/file system appropriately.

One aspect of the ASE I/O subsystem that is often misunderstood is the '*i/o batch size*' configuration parameter. Changing this can indirectly improve performance but generally has no relationship with individual user tasks except those involved in bulk row modifications (select/into, etc.). ASE monitors processes – and if the number of physical writes in non log devices exceeds the '*i/o batch size*' configuration setting, the task is put to sleep. By default, then, this will mostly apply to checkpoint and housekeeper tasks – with the notable exceptions of dump/load database as database dumps and loads are submitted via the Backup Server and sybmultibuf processes. However, increasing 'i/o batch size' may benefit processes performing select/into and other bulk row modifications.

Checkpoint & Housekeeper Tuning

Why is there a checkpoint process anyhow?? The first answer we'd likely hear is "to reduce recovery time". While this is true, it also can help improve scalability by eliminating blocking I/O calls and cache stalls. Note the term "blocking". The context here is that if each individual process was responsible for its own physical writes, every process would "block" until the write completed. For an OLTP system that is heavily dependent on write throughput, this would significantly reduce scalability. Further, it improves I/O efficiency by reducing the number of I/O's per page. Consider a typical transaction history table in which records are appended on the end. If each page contained 10 rows and every user task was responsible for its own I/O:

- Each individual task be slower as it awaited the physical write to return – not only for the data page, but each index leaf and intermediate node as required as well.
- Every other task attempting to write to that page would also have to block and wait for the I/O to complete (remember the problem of DMA access writes and the need for the MASS bit?)
- I/O efficiency would suffer as each page might get written to disk ten times – which would cause the I/O subsystem to be flooded.

Is the checkpoint necessary – isn't this what the wash marker is all about? Sort of. It is true that in managing the MRU→LRU chain, once a dirty page reaches the wash marker, it will have a physical I/O scheduled by the engine – asynchronously from the task that modified the page. However, in a very high volume system or a small cache, this can easily result in cache stalls – in which a physical read has to wait for the write to complete so that it can reuse the buffer. Consequently, it is the checkpoint and the housekeeper that helps prevent cache stalls by submitting I/O's in advance – not only helping speed recovery times, but also decreasing the potential for cache stalls.

On the opposite end of the spectrum, tuning the housekeeper too aggressively or leaving the '*recovery interval in minutes*' too low can impeded scalability by causing contention on the MASS bit with user tasks trying to modify data pages in memory. So, while the primary reason for the checkpoint processes and housekeeper tasks is to reduce recovery times, they indirectly can contribute to scalability by reducing the potential for cache stalls – but can also impede scalability if tuned too aggressively.

Checkpoint Tuning

Checkpoint tuning is a balance between recoverability, physical read performance and I/O throughput. Consider the following known facts about checkpoints:

- Every 60 seconds, the checkpoint process wakes up and tries to checkpoint each database.
- The checkpoint process estimates 6,000 rows per minute for recovery. If the number of log records in each database divided by 6,000 exceeds the configuration for '*recovery interval in minutes*', the server attempts to flush all the dirty pages in cache for that database. This happens irrespective of the database setting '*truncate log on checkpoint*'.

- The checkpoint will flush pages in '*i/o batch size*' groups of pages. Once that number of writes have been issued, the checkpoint process will go to sleep until they complete. Once completed, the checkpoint process is re-awoken and issues the next batch of writes.
- The time it takes to perform the checkpoint is dependent on how many pages are dirty as well as how long the checkpoint process may have to wait before acquiring the MASS bit on the MASS containing the pages and of course how fast the disks are.
- If there are no dirty pages (thanks to the housekeeper), the checkpoint is considered a 'free checkpoint'
- When the checkpoint completes, it writes a checkpoint record in the log as well as writes a list of pages flushed to disk using Page Flush Time Stamp (PFTS) records.

Finding the balance with the checkpoint process requires the following consideration:

- Increasing the '*recovery time in minutes*' decreases how often the checkpoint is actually run. However, this leaves more dirty pages in cache, which can lead to cache stalls if the wash marker initiated I/Os do not complete in a timely manner.
- Increasing the '*i/o batch size*' may decrease the time that a checkpoint takes. However, increasing the '*i/o batch size*' can also increase the spike of contention on MASS bits as well as the I/O load on the system by consuming disk i/o structures and other system resources.
- Increasing the '*number of checkpoint processes*' may be required if there are a lot of databases in the system. However, as each checkpoint process will be issuing '*i/o batch size*' number of I/O's, again, care must be taken to not exhaust the number of disk i/o structures.
- Increasing the '*housekeeper free write percent*' can decrease the number of I/O's required by the checkpoint process, reducing the spike by leveling the I/O's required over time. This may also result in "free checkpoints".

Consider the following sp_sysmon report fragment from a 50 engine ASE:

```
===============================================================================
      Sybase Adaptive Server Enterprise System Performance Report
===============================================================================

Server Version:        Adaptive Server Enterprise/12.5.3/EBF 13325 ESD#7/P/Sun_
Server Name:           [REDACTED]
Run Date:              Dec 10, 2009
Sampling Started at:   Dec 10, 2009 10:00:45
Sampling Ended at:     Dec 10, 2009 10:30:48
Sample Interval:       00:30:03
Sample Mode:           Reset Counters

===============================================================================

…

Metadata Cache Management
-------------------------

  Metadata Cache Summary         per sec      per xact       count  % of total
  -------------------------  ------------  ------------  ----------  ----------
…
  Open Database Usage
```

```
    Active                          n/a           n/a          123       n/a
    Max Ever Used Since Boot        n/a           n/a          123       n/a
    Free                            n/a           n/a           77       n/a
    Reuse Requests
      Succeeded                     n/a           n/a            0       n/a
      Failed                        n/a           n/a            0       n/a

…

Recovery Management
-------------------

  Checkpoints                    per sec      per xact       count  % of total
  -------------------------  ------------  ------------  ----------  ----------
    # of Normal Checkpoints           1.2           0.0        2165     100.0 %
    # of Free Checkpoints             0.0           0.0           0       0.0 %
  -------------------------  ------------  ------------  ----------
  Total Checkpoints                   1.2           0.0        2165

  Avg Time per Normal Chkpt       0.43279 seconds
```

Knowing that a checkpoint is initiated every 60 seconds, with 30 minutes * 123 databases, we have a possible total between 3,567 and 3,690 checkpoints (depending on boundary). However, the system only completed 2,169 – meaning fortunately that either some did not need checkpoints or each checkpoint was taking too long. To determine which case, we notice that each checkpoint was taking 0.433 seconds on average. With 123 databases active, that means it takes ~53 seconds for the checkpoint process to complete all the databases and start over. This is extremely close to the 1 minute checkpoint interval. While the sp_sysmon does not report the number of checkpoint processes, it is obvious that if a single checkpoint process is involved, it is steadily issuing writes. If the number of databases that required a checkpoint increased, the checkpoint would start lagging. Even more telling is that there were 0 “free checkpoints”.

What impact does this have on scalability? The answer is a bit indirect. As more servers become consolidated and the number of engines grow, not tuning the checkpoint process could lead to situations where more dirty pages are entering the wash area possibly leading to cache stalls. This can double the time for a physical read as it first must wait for the write to complete before the buffer is freed to be used for the read operation.

Housekeeper Tuning

There are three different housekeeper tasks in ASE:

Housekeeper Wash – focuses on flushing dirty pages to disk as I/O limits allow – irrespective of whether DOL or APL page. This task only runs at idle and is directly controlled by the ‘*housekeeper free write percent*’ configuration value. Setting this value to 0 disables the HK wash.

Housekeeper Garbage Collection – focuses on removing deallocated pages and deleted row space recovery for DOL (datarows and datapages). This task is critical but often untuned for applications that use DOL tables as it manages whether lazy (default) or aggressive (necessary for DOL) garbage collection is used.

Housekeeper Chores – this task also primarily runs at idle times and is responsible for flushing table statistics (forward rows, delete rows, empty pages – see systabstats…not index statistics), account statistics (CPU and I/O accounting), DTM detach timeouts and license usage information.

Output in sp_sysmon can be divided among the three. Consider the following sp_sysmon fragment (line numbers added for clarity):

```
1       Housekeeper Task Activity
2       -------------------------
3                                        per sec      per xact       count  % of total
4                                     ------------  ------------  ----------
5       Buffer Cache Washes
6         Clean                            127.9           1.3      383593      64.8 %
7         Dirty                             69.5           0.7      208472      35.2 %
8                                     ------------  ------------  ----------
9       Total Washes                       197.4           1.9      592065
10
11      Garbage Collections                  4.9           0.0       14766       n/a
12      Pages Processed in GC                0.6           0.0        1765       n/a
13
14      Statistics Updates                  13.0           0.1       38962       n/a
```

Lines 5-9 refer to the housekeeper wash, lines 11 & 12 refer to the housekeeper garbage collection, and the last line refers to housekeeper chores.

Housekeeper Wash

An important factor to keep in mind is that the housekeeper wash does not process 'logonly' or 'relaxed' caches – so the focus of the housekeeper wash is on 'mixed' caches that are using the normal MRU→LRU chain.

If the checkpoint process is taking too long and the number of free checkpoints is low, one alternative to increasing the number of checkpoint processes or lengthening the recovery interval is to instead tune the housekeeper wash. The housekeeper wash task is actually controlled by several different configurations.

First, the '*housekeeper free write percent*' configuration value controls how much of the I/O capacity can be used by the housekeeper wash. Note that this task is already at 'LOW' priority, so the inference is that this task will only be using the I/O capacity when the system isn't as busy or on a very low priority basis. The default value for '*housekeeper free write percent*' is only 1 – which is fine for a default installation, but fairly restricts this process otherwise. However, setting the value higher causes more contention for the MASS bits, blocking other processes from modifying pages in memory. Unfortunately, you cannot tell from the MDA Wait Events (31 & 52) which system process it is between the checkpoint and the housekeeper wash which one is actually blocking users. However, you might be able to infer which is causing the problems by looking at the number of PhysicalWrites in monProcessActivity. On most systems with the default '*recovery interval in minutes*' it is more likely that the checkpoint process is causing the contention. Before disabling the housekeeper wash (as is stated can be done in the docs), it likely is better first to prove if it isn't the checkpoint task – which runs at a higher priority – causing the problem vs. the housekeeper wash. Before increasing the 'housekeeper free write percent', you may want to baseline the amount of MASS contention due to writes – especially for automated tasks (message queues) or batch processes. Then, increase it slightly and compare the two. Generally, it should only be increase 1 or 2 percent at a time and special care should be taken before increasing over 10%.

The other method of controlling this task is dbcc tune(deviochar). In addition to the '*housekeeper free write percent*' configuration setting, the housekeeper wash has a default internal limit of only submitting 3 I/O's to any particular device. Consequently, if the number of free checkpoints is low, rather than increasing the '*housekeeper free write percent*', you may get better results by instead raising this limit – especially if most of the I/O is centered on a few

fast devices. Changing dbcc tune(deviochar) is not exactly a complete 1:1 with devices, however. What happens is this:

- The housekeeper wash flushes up to an '*i/o batch size*' of pages to disk.
- If flushing the pages results in more than deviochar (3) pages to any one device, the housekeeper wash moves to the next buffer pool in the cache (i.e. from 2K to 16K). If not, the housekeeper attempts to flush the next '*i/o batch size*' from the same buffer pool.
- If the pages to be flushed in the next pool affect the same device(s) already impacted by the deviochar limit, the housekeeper wash goes to the next pool without submitting any I/O's.
- If the housekeeper process finishes writing all dirty pages in all caches to disk, it checks the number of rows in the transaction log since the last checkpoint. If there are more than 100 log records, it issues a checkpoint. This is called a "free checkpoint".
- Once the housekeeper processes the last buffer pool, it goes to sleep.

The reality is that unless transactions are widely scattered across a lot of devices, the deviochar limit is likely going to be met within each buffer pool. Additionally, with the trend of fewer/larger devices vs. the more/smaller devices, it is likely that even different tables and databases within different caches and buffer pools will cause the limit to be reached. Even more restrictive is that most systems only have 2-3 named caches (including the default data cache) with a total of ~6 different buffer pools. As a result, the housekeeper wash is likely only going to be submitting one or very few '*i/o batch size*' group of records before going to sleep. As documented, the deviochar was originally based on slow devices – customers with SANs, SSDs/flash drives, or caching controllers may wish to increase this as a means of reducing the checkpoint time. As illustrated above in the topic on 'max async i/os per engine/server', the number of I/Os outstanding per device is often quite higher today than the ASE default settings. It can be increased on a per device or across all the devices within the system. For example:

```
-- dbcc tune(deviochar, <vdevno>, "<iolimit>")
-- tune housekeeper wash for SAN hosted system by increasing deviochar
-- across all devices to a new default of 20
dbcc tune(deviochar,-1,"20")  -- -1 for vdevno = all devices
go

-- increase the deviochar for specific devices such as those based
-- on SSDs still further
select name,vdevno from master..sysdevices
go
dbcc tune(deviochar,14,"50")
go
```

Note that dbcc tune() is not persistent – if the database instance is rebooted, you will need to re-issue this command. Additionally, there is no way to retrieve what the current "tuned" value is. Lastly, remember, that this is allowing the housekeeper wash task to submit more I/O than by default. While this can decrease the checkpoint time thus reducing the spike caused by checkpoints, the negative affect is that this may allow the housekeeper wash to submit multiple '*i/o batch size*' writes concurrently – increasing the MASS contention as mentioned earlier. Consequently, while increasing this may help housekeeper wash performance and reduce checkpoint times, it may also decrease overall system throughput. As with '*housekeeper free write percent*', you should baseline and measure the impact of changes.

A second way of controlling contention is by having the housekeeper skip caches that don't need to be flushed to disk. This avoids unnecessary contention on the MASS bit between the housekeeper and user tasks as well as allows the housekeeper to focus on just the caches it should. Consider the following example:

```
[Named Cache:tempdb_cache]
        cache size = 512M
        cache status = mixed cache
        cache status = HK ignore cache
        cache replacement policy = DEFAULT
        local cache partition number = DEFAULT
```

As documented in the ASE 15.0 manuals, the 'HK ignore cache' setting tells the housekeeper wash task to skip the entire cache (all buffer pools). This does **not** mean that writes will not happen. Remember, the cache's wash marker as well as the checkpoint process will still flush a dirty buffer to disk. However, this can be particularly useful in reducing the contention for MASS bits on caches that may not need to be worried about recoverability – or the recoverability can be controlled via a longer '*recovery interval in minutes*' setting. Consider the following:

Cache Contents	Comments on 'HK ignore cache'
tempdb tables	Ideally, we are trying to avoid any physical IO here as much as we can. Especially as tempdb is non-recoverable, having the housekeeper running in a dedicated cache for tempdb is just overhead.
Transaction Log	Because the log flush is required as part of the commit process for write-ahead logging, a 'log only' cache should have no dirty pages and therefore has no need for the housekeeper to run in that cache. Therefore the housekeeper task does not process 'log only' caches and there is no need to set the 'HK ignore cache' setting.
Indexes	This takes a bit of thought, but often index pages can be the most volatile – especially indexes on columns such as update datetime, etc. Due to the number of rows per page being changed, by delaying the physical write until either the checkpoint is reached or the wash marker is hit reduces the amount of I/O's that occur on index pages.
Text/Image data	Typically, writes to text/image data are minimally logged, consequently, the checkpoint process may not kick in for larger text/image column writes as quickly as if they were logged as it would if the same data was inserted in multiple rows. By disabling the HK for a cache dedicated to text/image, we ensure that the writes will be delayed until the wash marker. While typically multiple writes to the same text page does not happen, the number of writes for a text block of data likely will trip the deviochar limit quicker for the housekeeper – especially as the text/image data is likely on the same device as the normal data row. By disabling the HK, we can avoid tripping the deviochar limit as soon, which allows the HK to focus on writing out the data rows.
Job queue/workflow queues (in database tables)	Similar to the volatile index situation, these tables typically are written to extensively and eventually the page is likely deallocated as all the pending jobs in the queue are deleted from the table. By skipping these caches for the HK wash, we may be able to fully delay the physical writes until the wash marker (or checkpoint process) at which point the page may be able to be deallocated (or already was by the HKGC).

Note that there are some restrictions when using this. The 'HK ignore cache' cannot be the only cache status, and must appear with one of the other cache statuses such as 'default data cache', 'mixed cache', or 'log only' statuses. This actually is a bit of a safety factor as this option has to be set manually. If creating a cache by hand via direct modification of the config file, you might forget the other cache status information. However, if created by sp_cacheconfig, the cache will already have one of those three statues – so for normal cases this warning is a mute point. Note that since the housekeeper wash task only processes non-log records, enabling this option on a 'log only' cache is a bit redundant. It also is not advisable to enable this on the default data cache as most activity will likely be happening there and that is where you likely

need the housekeeper if anywhere. Consider the following table summarizing when the housekeeper wash runs:

Cache Status/Configuration	HK Wash?
'log only' cache status	No
'relaxed' cache strategy	No
'HK ignore cache' cache strategy	No
(all other conditions)	Yes

Another tuning tip that sometimes may apply is that you can raise the priority of the housekeeper wash task. Earlier, it was mentioned that this task normally runs at LOW priority. As will be discussed later, the procedure sp_setpsexe to change the priority from LOW (EC3) to MEDIUM (EC2) or HIGH (EC1). The danger in this is that it might increase the MASS bit contention – but it still could be useful if the housekeeper is not getting enough CPU time and the checkpoint process is still taking too long.

To summarize the above, you can increase the housekeeper wash performance through a variety of means:

- Increase the 'housekeeper free write percent' configuration setting
- Increase the dbcc tune(deviochar) (not persistent)
- Have the housekeeper skip unnecessary caches with 'HK ignore cache'
- Increase the process priority via sp_setpsexe (not persistent)

If monitoring the housekeeper wash monProcessActivity.PhysicalWrites suggests that it might be the primary cause of MASS bit contention, doing the opposite can benefit at the expense of longer checkpoints.

Housekeeper Garbage Collection

In addition to the checkpoint process's '*recovery interval in minutes*' often being mistuned, the next most frequently mistuned parameter is the '*enable housekeeper GC*' configuration parameter. The housekeeper GC task is responsible for reclaiming space from deleted rows and deallocating empty pages from DOL tables – both datarows and datapage locking.

As a bit of a background, in an APL locked table, since the entire page is locked – whether index or data page – when a change is being made, the task that is performing the delete also reclaims the space on the page. This is accomplished by consolidating any empty space on the page and then writing the new compacted page image to memory. If the deleted row was the last row on a page, then the task is responsible for deallocating the page. One side effect of this is that deletes (and deferred updates) can take longer as the space reclamation and page deallocation are part of the transactional context of the DML operation. Consider that each update of an index key value is a deferred update, and you can get a picture of how much overhead is spent performing space management within a normal update or delete operation.

With DOL locking, things are a bit different. While the latch does protect the page from being modified simultaneously, the transactions need not have committed for the next process needing to modify the page. Since the transactions have not committed and may be rolled back, any empty space cannot be reclaimed as this would require a lock escalation in nearly every instance to a page lock. As a result, the index or data row is simply marked for deletion, and the space reclamation is done by the housekeeper garbage collection process in ASE

asynchronously to the transaction. If the space can be reused either in place or after the page is compacted by the housekeeper GC, it will be - but if the last row is deleted, rather than the transaction being held up - the page is not reclaimed until the housekeeper gets around to it.

Now this is again where some ASE default server config defaults need to be understood in the context of other configuration settings. First, remember that the server's 'lock scheme' default value is 'allpages'. Then consider the following paragraph - which is a direct quote from the ASE product manuals - specifically the 12.5.1 System Admin Guide (vol 1), which stated under the section on config parameters for 'enable housekeeper GC':

> To configure Adaptive Server for garbage collection task, use:
>
> sp_configure 'enable housekeeper GC', value
>
> For example, enter:
>
> sp_configure 'enable housekeeper GC', 4
>
> The following are the valid values for enable housekeeper GC configuration parameter:
>
> - 0 - disables the housekeeper garbage collection task, but enables lazy garbage collection by the delete command. You must use reorg reclaim_space to deallocate empty pages. This is the cheapest option with the lowest performance impact, but it may cause performance problems if many empty pages accumulate. Sybase does not recommend using this value.
> - 1 - enables lazy garbage collection, by both the housekeeper garbage collection task and the delete command. This is the default value. If more empty pages accumulate than your application allows, consider options 4 or 5. You can use the optdiag utility to obtain statistics of empty pages.
> - 2 - reserved for future use.
> - 3 - reserved for future use.
> - 4 - enables aggressive garbage collection for both the housekeeper garbage collection task and the delete command. This option is the most effective, but the delete command is the most expensive. This option is ideal if the deletes on your DOL tables are in a batch.
> - 5 - enables aggressive garbage collection for the housekeeper, and lazy garbage collection by delete. This option is less expensive for deletes than option 4. This option is suitable when deletes are caused by concurrent transactions.
>
> **Using the reorg command**
>
> Garbage collection is most effective when you set enable housekeeper GC to 4 or 5. ***Sybase recommends that you set the parameter value to 5.*** [bold and italics authors] However, if performance considerations prevent setting this parameter to 4 or 5, and you have an accumulation of empty pages, run reorg on the affected tables. You can obtain statistics on empty pages through the optdiag utility.
>
> When the server is shut down or crashes, requests to deallocate pages that the housekeeper garbage collection task has not yet serviced are lost. These pages, empty but not deallocated by the housekeeper garbage collection task remain allocated until you remove them by running reorg.

Now that is straight from the ASE 12.5.1 manuals circa 2003….~7 years ago, so the tuning aspects have been documented for quite some time. Essentially, what it is pointing out is that if you use DOL locking anywhere in your application and you do not have the 'enable housekeeper GC' set to 4 or 5, it is likely that you will need to run the reorg command to reclaim the space. The question is which setting – 4 or 5 – is the most appropriate.

The answer is that if doing the usual batch purge jobs at night, the answer is to set the 'enable housekeeper GC' configuration to 4, while 5 would be more appropriate for deletes done during normal OLTP. Unfortunately, in most true OLTP systems, both batch and OLTP deletes happen – but on different tables. For example a batch process likely purges or archives older rows. Meanwhile, during normal processing, either deferred updates leave space – or deletes on volatile tables such as workflow queues implemented in SQL tables are occurring.

The question often is "What is meant by deletes being more expensive in setting 4 and 5?" The answer should be almost expected by now. In order to reclaim the space on a page, the housekeeper GC has to prevent other user tasks from trying to *read* or *write* to the page. The easiest way to do this, of course, is to grab the MASS bit as if it was going to do a physical write

of the page to disk – which it may have to do if placement is to be maintained – especially on indexes (more on this later).

As mentioned earlier, the default is because the default locking scheme is APL and nothing more than 1 is necessary if all the tables are APL. The original 12.5.0.3 white paper *Housekeeper Enhancements in Adaptive Server Enterprise 12.5.0.3* provided a bit more detail stating for the setting of 1 that:

> • 1 - Enables lazy garbage collection by both the housekeeper garbage collection task and the **delete** command. This is the default value. Empty pages may not be processed due to non-optimal lazy evaluation when there are long running transactions. This slowdown in empty page processing could result in ***HK GC queue overflow***,[bold italics authors] causing empty pages. If more empty pages accumulate than your application allows, consider using a value of 4 or 5 (described below). You can use the **optdiag** utility to obtain statistics on empty pages.

Before we get to the HK GC queue overflow that bolded and italicized above (or the shutdown/crash comments), let's take a look at optdiag. Here is a sample output:

```
Data page count:                 3065907
Empty data page count:           8
Data row count:                  5990014.0000000000000000
Forwarded row count:             32418.0000000000000000
Deleted row count:               35312.0000000000000000
Data page CR count:              383495.0000000000000000
OAM + allocation page count:     76852
First extent data pages:         89170
Data row size:                   898.3669699918971219
```

Ouch. 32,000 forwarded rows and 35,000 deleted rows. What happened - why is a reorg necessary?? Shouldn't the housekeeper GC prevented this? The answer is that the housekeeper GC process maintains a non-tunable queue of pending requests such as deletes. If the housekeeper GC is not running aggressively enough, this queue overflows and some of the requests will be "dropped" or lost - this is known as a "HK GC queue overflow". Worse yet, if you shutdown the server (or it crashes), since the HK GC queue is an in-memory only queue, the entire queue is lost - and the only way to reclaim all this "lost space" in either case is to run a reorg.

Of course, a ounce of prevention is always better than a pound of cure. In ASE 15.0.2 and higher, the MDA tables monEngine, monOpenObjectActivity and monOpenPartitionActivity track the number of HK GC queue overflows via columns in the schema such as:

```
create existing table monEngine (
        EngineNumber                    smallint,
        InstanceID                      tinyint,
        CurrentKPID                     int,
        PreviousKPID                    int,
        CPUTime                         int,
        SystemCPUTime                   int,
        UserCPUTime                     int,
        IOCPUTime                       int,
        IdleCPUTime                     int,
        Yields                          int,
        Connections                     int,
        DiskIOChecks                    int,
        DiskIOPolled                    int,
        DiskIOCompleted                 int,
        MaxOutstandingIOs               int,
        ProcessesAffinitied             int,
        ContextSwitches                 int,
        HkgcMaxQSize                    int,
        HkgcPendingItems                int,
        HkgcHWMItems                    int,
        HkgcOverflows                   int,
        Status                          varchar(20) NULL,
        StartTime                       datetime NULL,
        StopTime                        datetime NULL,
        AffinitiedToCPU                 int NULL,
        OSPID                           int NULL,
)
external procedure
at "@SERVER@...$monEngine"
```

```
go

create existing table monOpenObjectActivity (
        DBID                            int,
        ObjectID                        int,
        IndexID                         int,
        InstanceID                      tinyint,
        DBName                          varchar(30) NULL,
        ObjectName                      varchar(30) NULL,
        LogicalReads                    int NULL,
        PhysicalReads                   int NULL,
        APFReads                        int NULL,
        PagesRead                       int NULL,
        PhysicalWrites                  int NULL,
        PagesWritten                    int NULL,
        RowsInserted                    int NULL,
        RowsDeleted                     int NULL,
        RowsUpdated                     int NULL,
        Operations                      int NULL,
        LockRequests                    int NULL,
        LockWaits                       int NULL,
        OptSelectCount                  int NULL,
        LastOptSelectDate               datetime NULL,
        UsedCount                       int NULL,
        LastUsedDate                    datetime NULL,
        HkgcRequests                    int NULL,
        HkgcPending                     int NULL,
        HkgcOverflows                   int NULL,
        PhysicalLocks                   int NULL,
        PhysicalLocksRetained           int NULL,
        PhysicalLocksRetainWaited       int NULL,
        PhysicalLocksDeadlocks          int NULL,
        PhysicalLocksWaited             int NULL,
        PhysicalLocksPageTransfer       int NULL,
        TransferReqWaited               int NULL,
        AvgPhysicalLockWaitTime         real NULL,
        AvgTransferReqWaitTime          real NULL,
        TotalServiceRequests            int NULL,
        PhysicalLocksDowngraded         int NULL,
        PagesTransferred                int NULL,
        ClusterPageWrites               int NULL,
        AvgServiceTime                  real NULL,
        AvgTimeWaitedOnLocalUsers       real NULL,
        AvgTransferSendWaitTime         real NULL,
        AvgIOServiceTime                real NULL,
        AvgDowngradeServiceTime         real NULL,
)
external procedure
at "@SERVER@...$monOpenObjectActivity"
go
```

Not only can we detect how many overflows actually occur, we can tell which *tables* they actually occur on….and therefore which ones we need to run a reorg on. And by a quick look at the HkgcPending - we can determine if we should wait a bit before shutting down. As mentioned, monOpenPartitionActivity includes the same columns - so you could further narrow it to specific partitions - which could reduce the need to run reorg. If it is the oldest partition - and the one you are likely going to drop in a day or two - why bother?

Housekeeper Chores

The housekeeper chores task is a very lightweight task that focuses on some miscellaneous housekeeping. Specific tasks include:

- Flushing tables statistics (see below)
- Flushing account statistics (see below)
- Handling timeout of detached transactions. This can be turned off using the configuration parameter '*dtm detach timeout period*'.
- Checking license usage. You can turn it off using the '*license information*' configuration parameter .

The table statistics mentioned are those tracked in systabstats or reported at the top for each table in optdiag – essentially, the number of forwarded rows, empty pages, row count, deleted rows, etc. It also flushes the number of rows and columns modified to sysstatistics. It does not flush index statistics, such as those created by update index statistics. Effectively, the housekeeper chores tasks automates the statistics flushed by the sp_flushstats stored procedure.

The account statistics flushing is an often overlooked and very minor task that often doesn't even need to be run. This refers to the CPU and I/O accounting statistics used for chargeback accounting. Very few organizations use this information any more – and often even DBA's ignore this information when reviewing system load, despite the fact it could be an interesting indicator of who is consuming the most system resources. The part that is overlooked about this, is that in order to record this information, the housekeeper chores must update the master..syslogins table – which puts a low but sustained rate of activity on the master database. If not careful, this can lead to transaction log full problems in master. While the documentation states that the housekeeper chores is not controlled by a single configuration value, what isn't mentioned but obvious above is that some of the chores are controlled by configuration parameters – and the account statistics flushing is one of those. Typically most ASE systems will have a configuration file with:

```
[SQL Server Administration]
…
        cpu accounting flush interval = DEFAULT
        i/o accounting flush interval = DEFAULT
…
        sysstatistics flush interval = DEFAULT
```

The default for CPU and I/O accounting is 200 and 1000 clock ticks respectively while the default for the sysstatistics flush is 0 minutes (disabled). Enabling sysstatistics flushing does not start another housekeeper process as the existing housekeeper chores process assumes this task. Additionally, sysstatistics flushing only impacts the recording of rows/columns modified and not the systabstats level of information. Essentially, this is the system equivalent to the datachange() system function. According to the documentation, the optimizer uses these statistics, most likely in adjusting the data and index cluster ratios to accommodate recent changes in data fragmentation.

Recommendations for ASE

Before attempting to tune the checkpoint or housekeeper processes, the best thing is to get a couple of sp_sysmon outputs along with MDA table samples. Earlier illustrations contained sp_sysmon data. Ignoring the housekeeper GC overflows aspect, the typical MDA tables for tuning the checkpoint and housekeeper tasks are the monProcess, monProcessActivity and monProcessWaits tables.

The monProcess table really simply is used to identify the SPID and process information for these tasks. Consider the following monProcess snippet from a mostly idle server:

SPID	Engine Number	Priority	Application	Command	Seconds Waiting	Wait EventID	EngineGroup Name	Execution Class
5	0	5		CHECKPOINT SLEEP	9	57		
6	0	7		HK WASH	41	61		
7	1	5		HK GC	1	61		
8	1	7		HK CHORES	14	61		
16	0	5	DBISQL	AWAITING COMMAND	10	250	ANYENGINE	EC2

First, notice that the priority of the checkpoint and housekeeper GC tasks are at 'medium' priority (5) the same as a normal user task running at medium/EC2 execution class – while the housekeeper wash and chores tasks are operating at idle (7 – a priority user tasks cannot be assigned to). In addition, at this point in time, the processes are spread across the two engines – like any other task, the checkpoint and housekeeper tasks can run on any engine by default. Now, consider the following output from monProcessActivity (note, it is spid 16 doing all the deletes):

SPID	CPU Time	WaitTime	Physical Reads	Logical Reads	Pages Read	Physical Writes	Pages Written
5	200	3431500	9	2137	9	8028	53619
6	300	3429200	0	0	0	9044	57092
7	0	3473100	0	2455	0	248	248
8	0	3434200	0	13	0	18	18
16	18000	3434100	4577	3228653	25507	45449	82920

When aligned with the monProcess output above, you can easily see that the checkpoint (spid 5) and the housekeeper wash (6) have been involved in the most writes with the split just about even at 53,619 and 57,092. Notice also that both tasks were doing large I/O's – with an average of 6.7 and 6.3 pages per write. Meanwhile the garbage collection and chores housekeeper tasks did little actual writes. Interestingly, these were primarily on the index on the test table – visible from the following monOpenObjectActivity fragment.

ObjectID	Index ID	Logical Reads	Physical Writes	Pages Written	Rows Inserted	Rows Deleted	Rows Updated	Hkgc Requests	Hkgc Pending	Hkgc Overflows
681050431	0	801933	7539	53662	714934	152786	0	0	0	0
681050431	2	460848	426	426	0	152786	0	226	0	0

Regardless, the result is that most of the MASS bit contention should be attributable to the checkpoint as well as the housekeeper wash. Consider the following monProcessWaits fragment (joined with monWaitEventInfo to decode the WaitEventID's).

SPID	KPID	Wait EventID	Waits	WaitTime	Description
5	720907	29	9	100	waiting for regular buffer read to complete
5	720907	31	234	1700	waiting for buf write to complete before writing
5	720907	36	2	0	waiting for MASS to finish writing before changing
5	720907	51	391	29000	waiting for last i/o on MASS to complete
5	720907	52	27	1000	waiting for i/o on MASS initated by another task
5	720907	55	55	400	wait for i/o to finish after writing last log page
5	720907	57	65	3397400	checkpoint process idle loop
5	720907	150	1	0	waiting for a lock
5	720907	157	1967	1800	wait for object to be returned to pool
5	720907	214	55	0	waiting on run queue after yield
5	720907	272	1	0	waiting for lock on ULC
6	786444	31	2595	35300	waiting for buf write to complete before writing
6	786444	51	7	0	waiting for last i/o on MASS to complete
6	786444	52	115	900	waiting for i/o on MASS initated by another task
6	786444	54	8	100	waiting for write of the last log page to complete
6	786444	55	17	0	wait for i/o to finish after writing last log page

SPID	KPID	Wait EventID	Waits	WaitTime	Description
6	786444	61	187	3388800	hk: pause for some time
6	786444	157	2104	1600	wait for object to be returned to pool
6	786444	214	204	200	waiting on run queue after yield
7	851981	31	1	0	waiting for buf write to complete before writing
7	**851981**	**36**	**235**	**700**	**waiting for MASS to finish writing before changing**
7	**851981**	**51**	**21**	**100**	**waiting for last i/o on MASS to complete**
7	**851981**	**52**	**2**	**0**	**waiting for i/o on MASS initated by another task**
7	851981	54	3	0	waiting for write of the last log page to complete
7	851981	55	227	1300	wait for i/o to finish after writing last log page
7	851981	61	696	3461600	hk: pause for some time
7	851981	214	2312	600	waiting on run queue after yield
7	851981	272	1	0	waiting for lock on ULC
8	**917518**	**51**	**5**	**0**	**waiting for last i/o on MASS to complete**
8	917518	55	11	0	wait for i/o to finish after writing last log page
8	917518	61	58	3432800	hk: pause for some time
8	917518	214	352	0	waiting on run queue after yield
16	1572888	29	607	9500	waiting for regular buffer read to complete
16	1572888	31	4	0	waiting for buf write to complete before writing
16	**1572888**	**51**	**84**	**22000**	**waiting for last i/o on MASS to complete**
16	**1572888**	**52**	**4**	**400**	**waiting for i/o on MASS initated by another task**
16	1572888	124	3953	5100	wait for mass read to finish when getting page
16	1572888	157	30770	27700	wait for object to be returned to pool
16	1572888	214	55	0	waiting on run queue after yield
16	1572888	250	225	3369100	waiting for incoming network data
16	1572888	251	328	100	waiting for network send to complete

The first question you might ask is why the checkpoint and housekeeper wash tasks would have MASS bit contention (WaitEventID=52). The answer is that they could be contending with one another – or they could be in contention with a user tasks such as SPID 16 (specifically the WaitEventIDs such as 51 which is when SPID 16 is waiting on an I/O it issued). Overall, though, there are very few waits caused by MASS bit contention above – with the housekeeper wash having the most (115). If this was the result from a production system, we might conclude:

- Increasing the '*recovery interval in minutes*' to reduce possible contention between the housekeeper wash and checkpoint.
- There really isn't much contention between the checkpoint/housekeeper and the user task.
- Housekeeper GC likely does not need to be adjusted as there were no overflows and only a few requests.

The result is that backing off the recovery interval slightly would encourage the housekeeper, increase free checkpoints and would not increase MASS bit contention.

Keep in mind that not all MASS bit contention is between the user tasks and system tasks. As noted above, the user task was waiting for some physical writes that it had initiated for one of several reasons. Other user tasks wanting to modify those same pages would have to wait just as if the checkpoint or housekeeper had initiated the tasks. One way to get a sense of whether it is the system tasks or the user tasks is to look at the number of writes in monProcessActivity as well as the amount of latch (WaitEventID=41) or lock contention (WaitEventID=150). If processes were trying to write to the same pages, likely there would be at least some considerable amount of logical contention at the same time.

Keeping all of this in perspective, the following is a list of configuration parameters and tuning considerations.

Configuration Parameter	Default	Recommendation	Tuning Considerations
cpu accounting flush interval	200	0	Rarely if ever used – just adds to overhead in the master database, needlessly.
i/o accounting flush interval	1000	0	Rarely if ever used – just adds to overhead in the master database, needlessly.
sysstatistics flush interval	0	10	
disable disk mirroring	1	1	Leave turned off – older systems may have Sybase disk mirroring enabled although not being used any longer. Having Sybase mirroring enabled – even if not using it – can increase contention on device semaphores.
i/o polling process count	10	(see comments)	Compare sp_sysmon or monEngine Disk IO Checks to Disk IO Completions. If the IO completions are >85% of the IO checks, consider reducing this value by 1 to improve individual application responsiveness. If more throughput across applications is desired, especially on larger memory systems or when using a significant number of SSD's, consider increasing in increments of 10 (but monitor IO wait times via monProcessWaits).
i/o batch size	100	(default)	Increase if sp_sysmon reports a considerable percentage of task context switches due to "Exceeding I/O batch size". Biggest impact is likely within tempdb due to select/into performance as well as checkpoint speeds and housekeeper throughput.
disk i/o structures	256	300-400 per engine at a minimum	Number of disk I/O structures per server – limits the number of concurrent I/O operations for the entire server.
max async i/os per engine		Same as above	Used by all platforms on a per engine basis. See earlier notes as to why to set to total disk I/O structures vs. the 300-400/engine otherwise it should be minimally the same as disk i/o structures.
max async i/os per server		Same as above	Only used by Solaris and Windows NT in addition to the above for OS interaction. Minimally, should be at least the same as disk i/o structures.
number of checkpoint processes	1	(see comments)	1 per write intensive database, not to exceed 1 per engine. Caution should be used when number of checkpoint processes exceed 30-40% of the number of engines on high SMP to prevent MASS bit contention as well as leveling the spike in load due to checkpoints.
recovery interval in minutes	5	500	Based on 6,000 rows/minute (120/second) which is a bit low considering today's system speeds. Result on a high volume system is that every checkpoint interval (1 minute) a full checkpoint is performed – especially if the system is performing >1,000 inserts/second. While 500 minutes seems like a long recovery time, keep in mind this is based 6,000 rows/minute - which means 3,000,000 log records – or 10 minutes of activity at 5,000 log records per second (1 data row plus 4 index rows @ 1,000 inserts/second). While recovery can take longer, adjusting the housekeeper can reduce this.
housekeeper free write percent	1	<=10	Even out the spikes caused by checkpoints and reduce the need for checkpoints.
enable housekeeper GC	1	5	It is a rare database that doesn't have at least one DOL table these days….so should be either 4 or 5.

Please note that these are only starting recommendations and tips – your requirements may differ.

Lock Manager Tuning

The notion of spinlocks – especially as it relates to the lock manager and object manager are also confusing aspects. The simple fact is that the list of open objects and list of active locks are very dynamic from the aspect of the DBMS engine – especially the list of locks. Since most modern DBMS's are multi-threaded/multi-user/multi-process applications, adding or removing an object descriptor or lock from the active list requires modifying memory. We all are aware that modifying memory in a multi-threaded or multi-process system requires protecting that resource with a mutex. In ASE, these mutexes are called "spinlocks" as processes waiting for the resource are not put to sleep, but continuously poll for mutex availability. By contrast, the log semaphore is not a spinlock as tasks waiting on it do get put to sleep. Of course, each mutex requires a bit of memory as well as latching mechanisms. If the system had a single mutex for every object or every page/row lock, the overhead would be high. In addition, with multiple mutexes, there has to be either a hash list that helps a task find the appropriate mutex it needs – or else the task has to do a serial scan of the list of mutexes. The fewer the mutexes, the faster it is to find the mutex in the list.

A second danger of large numbers of mutexes is the risk that two different user tasks will be holding one mutex (i.e. modifying a lock chain) while needing another mutex. This can lead to internal deadlocks – and is a problem that plagues Oracle systems in high volume situations due to ITL deadlocks.

Lock Hashtable & Spinlocks

This has considerable applicability to the ASE lock manager due to the way the lock manager works. Locking in Sybase is implemented using a lock hashtable and a simple serial lock chain. To see how this works, assume that we start with 10 hash buckets and the hash function is a basic mod() function. Additionally, assume our spinlock ratio is 3 – which for 10 hash buckets means a total of 4 spinlocks (the last one only guarding one hash bucket). If we want an exclusive lock on page 25, 25%10=5 - so we would simply do a linear scan of the lock chain in bucket 5. If we find a conflicting lock, we know we have to block. However, in any case, we need to add our lock to the lock chain, so we grab the spinlock on the lock hash bucket, modify the lock chain, and release the spinlock. This is illustrated below:

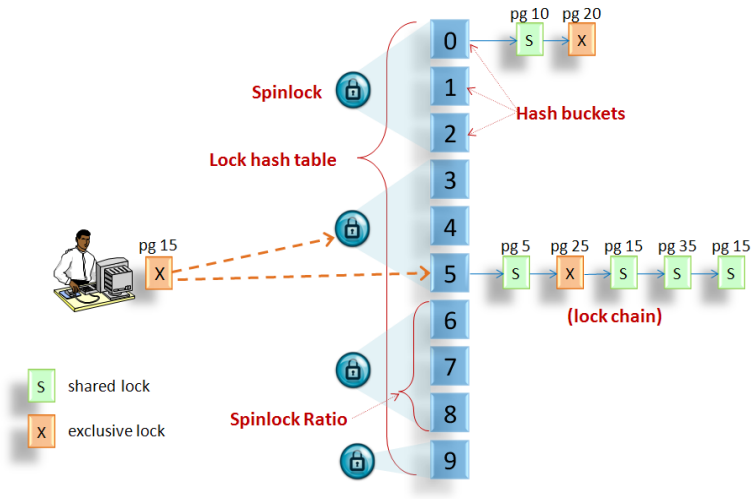


Figure 4 – Lock Hashtable, Lock Chains & Spinlock Ratios

The problem of course is that if the lock chain is long, the serial scan can take too long comparatively. For example, if I have 100,000 locks and only 10 hash buckets, each bucket would have 10,000 locks that I would have to check to see if there was a conflict. Obviously, not ideal. Consequently, Sybase ASE starts with a default 'lock hashtable size' of 2048 for row and page locks. A quick check of Sybase's P&T guide will also state that you should never let the lock chain length get greater than 5 - which is visible via sp_sysmon from the section:

```
Lock Management
---------------
…
  Lock Detail                    per sec      per xact       count  % of total
  -------------------------  ------------  ------------  ----------  ----------

  Table Lock Hashtable
    Lookups                      15177.9          13.2     1821353       n/a
    Avg Chain Length                 n/a           n/a     2.11600       n/a
    Spinlock Contention              n/a           n/a         n/a       0.0 %
…
  Page & Row Lock HashTable
    Lookups                     266115.3         232.2    31933831       n/a
    Avg Chain Length                 n/a           n/a     0.04418       n/a
    Spinlock Contention              n/a           n/a         n/a       0.0 %
```

We see that ASE actually maintains not one but two hashtables - one for table level locks and one for page & row locks. Now this is where a tad bit of tuning advice comes in handy. If this was the peak period for this system, likely no tuning for the lock manager is really necessary – although we need to keep in mind that this is the average across all the sp_sysmon time and not the peak. The above shows an average chain length of less than 1 for each page or row lock. But if that crept up to 3 or 4 or higher, that might be the point to start thinking about increasing the '*lock hashtable size*' and '*lock table spinlock ratio*' server configuration parameters.

It is sometimes confusing, but the *table* locks (vs. page/row)are maintained in a separate form of hash table which is controlled by the '*lock table spinlock ratio*' - in which the server uses an non-tunable 101 hash buckets to track all the table level locks. With a default of 20 for the '*lock table spinlock ratio*', this means that each spinlock is managing 6 table lock hash buckets. While it might seem odd in the above that the table lock hash chains are longer than the page/row hash chains, the fact there is fewer of them coupled with the fact that #temp tables typically cause table locks, it is easily explainable.

But what if it was the Page/Row hashtable that was showing a longer chain length? The page/row lock hashtable is controlled by the server configuration '*lock hashtable size*' for pure SMP systems and autotuned in later ASE Cluster Edition versions. Unlike table level lock hashtables, you can tune both the hashtable size and the spinlock contention. In the above example, the spinlock contention is effectively non-existent. But if there was any at all - even 1 or 2% - that is when you start thinking about changing the server configuration for '*lock spinlock ratio*'. By default, ASE starts with a '*lock spinlock ratio*' of 85 - which means a single spinlock guards 85 of the hash buckets - those 2048 buckets in the '*lock hashtable size*'…..which some quick math (or reading the manuals) points out that by default there are only 26 spinlocks guarding all the hash buckets for ASE. So if you start to see spinlock contention - simply *decreasing* the '*lock spinlock ratio*' to say 32 means that instead of 26 spinlocks, you would have 64 spinlocks - not a huge number - but it can make a huge difference if there is contention. Another way to increase the number of spinlocks is to increase the '*lock hashtable size*'. If you think about it, ASE starts with a default of 10,000 locks - which with ~2,000 hash buckets would result in an average of 5 locks per lock chain if evenly distributed. So, if we tune the server to increase the number of locks to something more realistic - such as 250,000 - if we leave the '*lock hashtable size*' at 2048 - we could easily hit a lock chain size of 125 if all the locks were in use and evenly distributed. That's 25 time larger than recommended. Keeping that ratio in mind - it might not be a bad idea to increase the '*lock hashtable size*' to '*number of locks*'/5 - or setting the '*lock hashtable size*' to 50,000 - which at the default '*lock spinlock ratio*' of 85 would result in ~600 spinlocks.

Address Locks & Optimistic Locking

In addition to page, row and table locks, ASE has address locks. Remember, for DOL tables, ASE uses a latch, consequently there are no locks used for the index level of a DOL table. APL tables use "address locks" essentially for the same purpose. In earlier times, ASE used a form of optimistic locking in which a DML query would traverse the index without locking it. Then if it had to modify the index key values, it would have to go back up the index chain acquiring locks – depending on how much index maintenance was also involved. ASE currently instead uses an address lock mechanism instead – implementing a form of pessimistic locking. As the DML query traverses down the index tree, it grabs address locks on the index pages as it progresses. Like other locks, an address lock can be shared or exclusive. Essentially, though, you should think of them as just like a normal page lock for an APL table for the following reasons:

- An exclusive address lock blocks access via that index not only to the data the DML statement is affecting, but all other rows the index page might point to. While a SPID could use a different index to get to the row, it still reflects index contention.
- When a leaf-level index page is exclusively locked, no modifications can be done to any data pointed to by the index rows on the locked page, since the index rows may need to be modified as well.

Each root page of an index for a DOL table has a spinlock guarding access to it. This can lead to spinlock contention even when there isn't address lock contention. There are two ways of dealing with the problem. The first is to re-enable optimistic locking. If the system is not experiencing a lot of address lock contention but is showing some address spinlock contention,

than optimistic locking may be a choice if the index is not very volatile. The reason for the caveat (index volatility) is that with optimistic locking, if the SPID needs to modify the index, it will restart the index traversal and use exclusive locks. This may cause an escalation to a table lock if the root page of the index needs to be changed. Address lock & spinlock contention is reported in sp_sysmon with a section similar to:

```
Task Context Switches Due To:
  Voluntary Yields              388.3            0.2       701586       1.7 %
  Cache Search Misses           788.5            0.3      1424849       3.5 %
  System Disk Writes             10.7            0.0        19399       0.0 %
  I/O Pacing                    411.1            0.2       742887       1.8 %
  Logical Lock Contention         1.9            0.0         3428       0.0 %
  Address Lock Contention         0.0            0.0           42       0.0 %
  Latch Contention                0.7            0.0         1229       0.0 %
  Log Semaphore Contention        4.0            0.0         7141       0.0 %
  PLC Lock Contention             1.4            0.0         2546       0.0 %
  Group Commit Sleeps            91.9            0.0       166078       0.4 %
  Last Log Page Writes          178.4            0.1       322455       0.8 %
  Modify Conflicts               31.1            0.0        56219       0.1 %
  I/O Device Contention           0.0            0.0            0       0.0 %
  Network Packet Received      5884.9            2.4     10634076      26.1 %
  Network Packet Sent          4482.5            1.8      8099891      19.9 %
  Other Causes                10261.2            4.2     18541939      45.5 %

…

Address Lock Hashtable
  Lookups                    150684.1           61.2    272286133       n/a
  Avg Chain Length                n/a            n/a      0.00129       n/a
  Spinlock Contention             n/a            n/a          n/a       1.0 %

Exclusive Address
  Granted                       1270.7            0.5      2296212     100.0 %
  Waited                          0.0            0.0            0       0.0 %
-------------------------  ------------  ------------  ----------  ----------
Total EX-Address Requests       1270.7            0.5      2296212       0.2 %

Shared Address
  Granted                     149413.4           60.6    269989940     100.0 %
  Waited                          0.0            0.0           42       0.0 %
-------------------------  ------------  ------------  ----------  ----------
Total SH-Address Requests     149413.4           60.6    269989982      25.4 %
```

The above shows negligible address lock contention, but is showing spinlock contention on the address lock hashtable. If there were address lock contention task switches with the same little or no address lock contention, enabling optimistic locking might be an option. Optimistic locking can be enabled by:

```
exec sp_chgattribute <tablename>, "optimistic_index_lock", [0 | 1]  -- 0=off, 1=on
```

In the above case, the contention is on the spinlock – and likely for the same reason as the normal lock manager spinlock reasons – the default spinlock ratio for address locks is 100 – which means 1 spinlock is guarding 100 root pages for different indexes. If there are approximately 5 indexes per table, this works out to 1 spinlock protecting all the index root pages for 20 tables – and hence the contention – even when there isn't address lock contention.

If there is address lock contention, the best option is to use DOL locking for the table.

DOL Tables & Tablescans

DOL tables have an interesting property that can improve scalability – the concurrency_opt_threshold – which by default is set to 15. The central idea is that tables with fewer pages than this setting will always use an index – which is sort of the opposite of APL tables and the 10 rows/10 pages rule. However, for tables with more than the

concurrency_opt_threshold, the optimizer may consider a tablescan over an index if the tablescan appears to use fewer I/O's. For example, consider a table with 1000 pages. If the table is used in a join or a low cardinality SARG is used, if the number of pages that need to be read are more than 40% of the table, a tablescan is considered as a means to reduce the I/O. The reason it works is that a table of 1000 pages will likely have at least 1 and possibly 2 levels of indexing – depending on the index key length. A query that needs access to the rows on 400 of the pages would need minimally 400 I/O's per index level plus the 400 I/O's for the data pages. By contrast, a tablescan would only need 1000 pages – a possible savings of 200 I/O's with 2 levels of indexing.

The problem is that this table scan can increase the contention and deadlocking within the table. It is possible that 60% of the table will need locks for which no data exists on that page or row to match the query criteria. This could be especially troublesome in situations where isolation level 3 is necessary and the locks are held for the duration of the transaction. This can be prevented on a table-by-table basis by increasing the concurrency_opt_threshold via:

```
exec sp_chgattribute <tablename>, " concurrency_opt_threshold", <value>
```

Before enabling this, you may want to observe the contention on the table and tablescans using monOpenObjectActivity as well as observing any deadlocks that happen using monDeadLock. If there are a considerable number of table scans and query plan analysis shows that it is ignoring the index due to the number of pages to be considered, you can set this value for the table to force queries to use indices instead.

Like all features, there is a negative beyond the increase in I/O from the index traversal aspect. Typically, due to the table scan, such small tables end up as the outer table in the joins. By forcing the query optimizer to choose an index over the fewer I/O's, the query join order may change – which could have a bigger impact on the query than the index traversals if the table involved is a key driving table for determining the results.

Lock Escalations and Scalability

One the other frequently mistuned ASE configuration parameters concerns lock escalations. Both row and page locks have configurable escalation configurations as follows:

Lock Escalation Low Water Mark (LWM) – When a user's task has fewer than the LWM number of locks, the server will not attempt to acquire a table lock.

Lock Escalation High Water Mark (HWM) – When a user's task has more than the HWM number of locks, the server will attempt to acquire a table lock.

Lock Escalation Percent (PCT) – This is often misunderstood. When the user's task has a number of locks that is between the LWM and the HWM, the server will compare the total pages in the table with the PCT value. If the PCT value exceeds the equivalent percentage of total pages in the table, a table lock will be attempted. For example, if the LWM=200 and the HWM=2000 and PCT=50 and the task currently has 450 locks, if the table only has 900 pages, the task will attempt to escalate to a table lock. If the table has 9 million pages, a table lock won't be attempted until the HWM is reached as 500 locks (up through 2000) is nowhere near 50% of the table.

The default values for the LWM and HWM are 200 each while the default value for the PCT is 100. Simply put, at the default configuration, once a process has acquired more than 200 page or row locks, it will try to grab a table lock – even if there are 2 billion rows in the table. Needless to say, if it succeeds in acquiring the table lock, other users are needlessly blocked from the table – crippling system scalability. This is particularly noteworthy as it is common that developers coding batch processes or other bulk modifications typically use logic similar to 'set rowcount 250' up through 'set rowcount 2000'. Often times, the only thing preventing a table lock is the fact some other process currently has an exclusive lock somewhere in the table – which forces the task trying to escalate to a table lock to continue with page or row locks instead.

Lock Manager Recommendations

Accordingly, the recommendations for ASE are as follows (based on the assumption of a large SMP server):

Configuration Parameter	Default	Recommendation	Tuning Considerations
number of locks	10000	(see comments)	Start with 10,000*max online engines*5
lock spinlock ratio	85	40 ≥ x ≥ 20	
lock address spinlock ratio	100	40 ≥ x ≥ 20	
lock table spinlock ratio	20	(see comments)	Decrease in steps to {20,15,10,7,5} if the lock chain length in sp_sysmon exceeds 3-4
lock hashtable size	2048	(see comments)	formula = number of locks / 5
lock wait period	2147483647	1800	Rather than waiting infinitely, abort after 30 minutes
page lock promotion HWM	200	2000	
page lock promotion LWM	200	200	
page lock promotion PCT	100	50	Only effective when current number of locks is between LWM and HWM
row lock promotion HWM	200	10000	
row lock promotion LWM	200	1000	
row lock promotion PCT	100	100	Only effective when current number of locks is between LWM and HWM

Again, please test your configuration and monitor to determine what is appropriate for your application.

Object Manager Tuning

One very crucial factor to understand: as the number of online engines increase, the amount of contention for spinlocks can increase exponentially. Nowhere is this more true than with the object manager. A good paper for better understanding the metadata management in ASE is the white paper titled *"Understanding The*

Metadata Cache" written in September 2002 by David Wein of Sybase ASE Engineering. Much of the information in this section is based on that paper as well as other discussions.

Metadata Descriptors

First, a bit of background. An "object" in ASE is essentially anything you would expect to find in sysobjects – tables, stored procedures, triggers, defaults/rules, etc. As a DBMS, ASE is not necessarily aware of the environment it will be deployed in – specifically how much memory

will be available. As a result, ASE keeps a dynamic list of which objects are *currently* being used. Obviously, this list can grow as well as shrink depending on the activity within the server, the available memory and the server configuration limits. These lists are often referred to as the meta-data cache or lists of descriptors as the meta-data cache contains more than just the object descriptors. Consider the following table of the descriptors in the meta-data cache:

Descriptor	Object	Config Parameter	Contents
DBTABLE	sysdatabases	number of open databases	sysdatabases row info; sysusages info (DISKMAP); Number of open contexts to the database (aka dbt_keep); timestamp (used for populating timestamp columns, etc.); db options; repagent secondary truncation point; pointer to DES; etc.
DES	sysobjects	number of open objects	sysobjects row info; (varies by object type – rest of this data assumes a table object) pointer to the IDES for table (indid=0 or 1); pointer to PDES for the table; systabstats info (indid=0 or 1); identity value information; cache bindings; lock promotion information; number of open contexts (dbkeepcnt); SQLDML replication status/threshold, spinlock location in memory, etc.
IDES	sysindexes	number of open indexes	sysindexes row info; systabstats row info for the index; cache bindings; number of open contexts (id_keepcnt), MDA monOpenObjectActivity counters, spinlock location in memory, etc.
PDES	syspartitions*	number of open partitions	syspartitions row info; etc.

** syspartitions as defined in ASE 15.x vs. 12.5.x*

An important aspect to keep in mind that the descriptor lists are caches – and will stay in memory until pushed out. If the number of descriptors is not enough, this descriptor reuse will occur – a very punitive process called "DES scavenging". When DES scavenging happens, the overhead is enormous, as ASE needs to:

- Write an error to the errorlog
- Find an available DES that isn't currently being used (keep count=0). If a DES can't be scavenged, the query has to be aborted and the client application receives an error about which object (index, etc.) could not be reused.
- Flush all the child object descriptors from cache. For a database, this means all object descriptors, index descriptors and partitions descriptors have to be removed from data cache.
- If a table object descriptor is re-used, any dirty pages must be first flushed to disk. A stored proc DES scavenge needs remove all copies of the proc from proc cache.
- Cleanup the memory structures and read system tables to populate the new DES

Note that an object descriptor can be anything in sysobjects. Hence when scavenging, ASE first checks for non-table objects. While this may prevent dirty pages from having to be written out first, it can lead to proc cache volatility. Re-use is best reported using sp_sysmon with a caveat about stored procedures:

```
Metadata Cache Management
-------------------------

  Metadata Cache Summary         per sec      per xact       count  % of total
  -------------------------  ------------  ------------  ----------  ----------
```

```
Open Object Usage
  Active                            n/a           n/a         5948       n/a
  Max Ever Used Since Boot          n/a           n/a        72863       n/a
  Free                              n/a           n/a        94052       n/a
  Reuse Requests
    Succeeded                       n/a           n/a            0       n/a
    Failed                          n/a           n/a            0       n/a

Open Index Usage
  Active                            n/a           n/a         6451       n/a
  Max Ever Used Since Boot          n/a           n/a        25745       n/a
  Free                              n/a           n/a        93549       n/a
  Reuse Requests
    Succeeded                       n/a           n/a            0       n/a
    Failed                          n/a           n/a            0       n/a

Open Database Usage
  Active                            n/a           n/a           71       n/a
  Max Ever Used Since Boot          n/a           n/a           71       n/a
  Free                              n/a           n/a          184       n/a
  Reuse Requests
    Succeeded                       n/a           n/a            0       n/a
    Failed                          n/a           n/a            0       n/a

Object Manager Spinlock Contention  n/a           n/a          n/a       9.3 %

Object Spinlock Contention          n/a           n/a          n/a       0.0 %

Index Spinlock Contention           n/a           n/a          n/a       0.0 %

Hash Spinlock Contention            n/a           n/a          n/a       0.0 %
```

The caveat about stored procedures is that concurrent executions of the same procedure results in another copy of the procedure (and possibly with a different plan) being loaded into proc cache. While, the DES for the procedure may not get reused, the proc cache volatility could still be causing a problem (more on this topic later in the section on Procedure Cache).

There is another way that scavenging can happen besides just if the number of descriptors is too low. If a child descriptor is not available, ASE may have to scavenge a higher level descriptor to free up descriptors at that level. For example, the default ASE is configured for 500 object and index descriptors. Assume we are simply referring to tables and that each table as 5 indexes. Since each index will need to be opened during query compilation, as soon as we reach 100 active tables (that remain active), all 500 of our index descriptors will be in use. The next index that is needed be opened will force an object descriptor to be scavenged to free up index descriptors.

Most DBA's are aware of how to use sp_monitorconfig 'all' to ensure that the number of descriptors does not exceed a maximum. For example (output is from a different server than the above):

```
1> sp_monitorconfig "all"
2> go
Usage information at date and time: Feb 11 2010  4:50PM.

 Name                      Num_free    Num_active  Pct_act Max_Used    Reuse_cnt
 ------------------------- ----------- ----------- ------- ----------- -----------
 additional network memory     5451712    28428352  83.91      28443936           0
 audit queue size                  100           0   0.00            22           0
 disk i/o structures              2048           0   0.00             0           0
 heap memory per user             4095           1   0.02             1           0
 max cis remote connection        4096           0   0.00             0           0
 max memory                     943084    33232788  97.24      33232788           0
 max number network listen           4           1  20.00             1           0
 max online engines                  0          20 100.00            20           0
 memory per worker process        9691         309   3.09           415           0
 number of alarms                   29          11  27.50            15           0
 number of aux scan descri        3000           0   0.00            22           0
 number of devices                 126          74  37.00            74           0
 number of dtx participant         500           0   0.00             0           0
 number of java sockets            562         479  46.01           487           0
```

number of large i/o buffe	8	0	0.00	1	0
number of locks	1998295	1705	0.09	166124	0
number of mailboxes	0	327	100.00	333	0
number of messages	64	0	0.00	0	0
number of open databases	11	14	56.00	14	0
number of open indexes	729	1271	63.55	1281	0
number of open objects	6111	3889	38.89	4360	0
number of open partitions	7124	2876	28.76	2886	0
number of remote connecti	350	0	0.00	2	0
number of remote logins	20	0	0.00	2	0
number of remote sites	10	0	0.00	1	0
number of sort buffers	19940	60	0.30	21370	0
number of user connection	546	478	46.68	509	0
number of worker processe	0	160	100.00	288	0
partition groups	1024	0	0.00	0	0
permission cache entries	906	94	9.40	94	90109
procedure cache size	1999456	544	0.03	259238	0
size of global fixed heap	299	1	0.33	1	0
size of process object he	3000	0	0.00	0	0
size of shared class heap	3072	0	0.00	0	0
size of unilib cache	373596	976	0.26	976	0
txn to pss ratio	16384	0	0.00	0	0

However, many may not be aware of just how dynamic the DES/IDES/PDES lists are. For example, every #temp table will need a DES/IDES/PDES created. A cached statement (ASE 15 statement caching) or a fully prepared statement both use Light Weight Procedures (LWP) – which also are objects (as any stored proc) and will need minimally a DES structure.

Object Manager Spinlock

Ignoring the DES scavenging problem, the most common problem with the descriptors is spinlock contention. Each descriptor contains information that is either highly volatile or semi-volatile. For example, the keep counts (highlighted in yellow above) are highly volatile – especially for tables that are frequently used in transactions. Other parts of the descriptor are less volatile and may only change with a schema modification or similar operation. However, even if not being changed, a descriptor may be accessed frequently to find child objects. For example, the DES is frequently accessed to determine the IDES or PDES locations. Although this sounds odd, if you think about it, in ASE 12.5, data storage was linked to sysindexes. In ASE 15, this changed to syspartitions – which can be thought of as another layer of nesting underneath sysindexes as an index (or table) can be partitioned. Consequently, all the access to the data for the table typically involves going to the DES first and then subsequently accessing the IDES(s) for that table to find out which indexes exist and the PDES(s) for each index to find the storage location of the data (DES → IDES → PDES).

This can pose a problem as each time a table is referenced in a query, the keep counts have to be incremented – not only for the tables, but also the index and partition descriptors. Changing these values requires grabbing the spinlock that is guarding the descriptors. There are multiple different types of spinlocks at the database level protecting different pieces of the DBTABLE, but due to the quantities, DES, IDES and PDES descriptors have a number of spinlocks governed by the configuration values. The question that arises is whether the object manager spinlock contention is due to concurrency on the same object descriptors or due to the object manager itself. The object manager is often depicted using a diagram similar to:

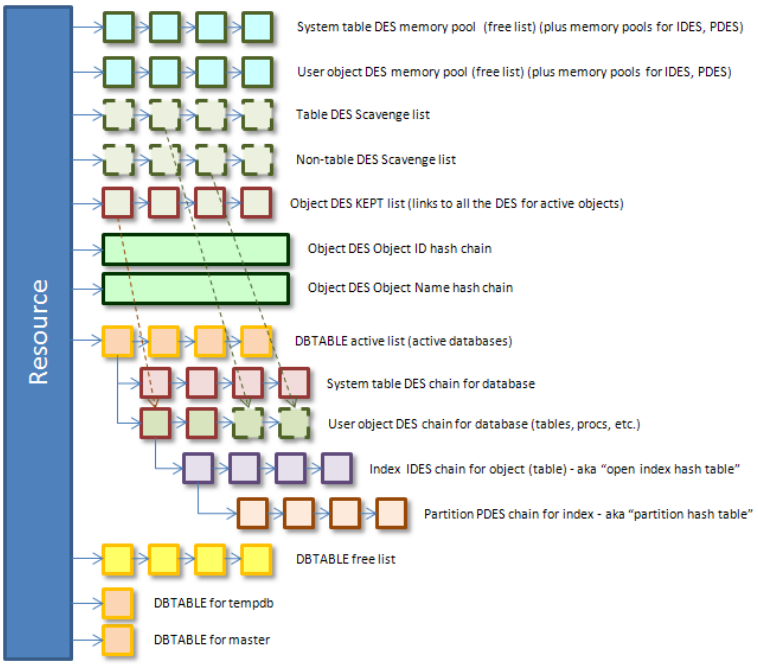


Figure 5 - ASE Object Manager Descriptors

The key focus above is the KEPT and Scavenge lists near the top of the diagram. Any DES that is active is linked onto the KEPT list. If the DES is not active, it is linked onto the appropriate "Scavenge" list. The Scavenge list operates like an MRU→LRU chain with recently appended object descriptors added to the MRU side – and any object reuse will use object descriptors from the LRU side of the list. This means that hot tables are constantly moving between the KEPT and Scavenge lists – note that the DES itself isn't moving (it is just an in-memory structure) – just it is being linked into the lists. Remember, though, the list will need to be maintained – and with concurrent users modifying this list, contention on the spinlock protecting the list can get quite high. This is reported as "Object Manager Spinlock Contention" (Resource->rdesmgr_spin if you use the sp_sysmon @dump_counters='Y' option). Note, that this is different from "Object Spinlock Contention" which can often be mitigated by decreasing the object spinlock ratio.

The easiest way to reduce "Object Manager Spinlock Contention" is to use dbcc tune(des_bind) as in:

```
-- bind an object's DES to the KEPT list
dbcc tune(des_bind, <dbid>, <objname>)
go

-- unbind an object's DES from the KEPT list
dbcc tune(des_unbind, <dbid>, <objname>)
go

-- find the bind state for objects that were bound in a database
-- used to verify if dbcc tune(des_bind) was run
dbcc tune(hotdes,<dbid>)
go
```

This binding to the KEPT list is not persistent across ASE reboots as the KEPT list is recreated from scratch with each server reboot. By preventing hot objects from being moved from the KEPT list, the amount of changes to the KEPT and Scavenge lists can be minimized, thus reducing contention on the "Object Manager Spinlock". Note that you can bind both tables and stored procedures (or any object). If binding a table that uses a default or a rule, you may also

wish to bind the rule and/or default. If the spinlock contention is more than 10%, you will also see the following at the bottom of the metadata section of the sp_sysmon output:

```
Metadata Cache Management
-------------------------

  Metadata Cache Summary         per sec      per xact       count  % of total
  -------------------------  ------------  ------------  ----------  ----------

…

  Object Manager Spinlock Contention  n/a           n/a         n/a       14.0 %

  Object Spinlock Contention          n/a           n/a         n/a        0.0 %

  Index Spinlock Contention           n/a           n/a         n/a        0.0 %

  Hash Spinlock Contention            n/a           n/a         n/a        0.0 %

  Tuning Recommendations for Metadata Cache Management
  ---------------------------------------------------
  - Consider identifying the hot objects using sp_object_stats
    and using dbcc tune(des_bind, <dbid>, <objname>) on the
    hottest set of objects.
```

Frequently, DBA's will focus on the "hot" tables – forgetting the "hot" procedures that may be being flushed out of cache due to procedure cache issues. Additionally, the tip above mentions sp_object_stats, which only reports when an object has locking contention – a highly accessed read-only table can cause problems with the Object Manager Spinlock as well. Consequently, you may want to instead look at monOpenObjectActivity.Operations column or the count(PlanID) from monCachedProcedures to determine which objects are likely candidates.

Recommendations

The following recommendations are starting guidelines for configuring ASE to avoid DES scavenging. The server should be monitored using sp_monitorconfig periodically to ensure proper tuning:

Configuration Parameter	Default	Recommendation	Tuning Considerations
number of open databases	12	(see comments)	count(*) from sysdatabases plus 10
number of open objects	500	5000	Includes tables, procs, rules, defaults, cached statements, etc. Monitor with sp_monitorconfig for reuse and increase by 10% if >85% active.
open object spinlock ratio	100	(100)	Rarely a problem – watch sp_sysmon for 'Object Spinlock Contention' (note: 'Object' – not 'Object Manager') and reduce by 50% if it occurs
number of open indexes	500	(see comments)	starting formula = (number of open objects/2)*4. Assumption is 50% of the active objects will be procs, rules, defaults – which don't have indexes. Monitor with sp_monitorconfig for reuse and increase by 10% if >85% active.
open index hash spinlock ratio	100	(100)	Rarely a problem – watch sp_sysmon for 'Hash Spinlock Contention' and reduce by 50% if it occurs.
open index spinlock ratio	100	(100)	Rarely a problem – watch sp_sysmon for 'Index Spinlock Contention' and reduce by 50% if it occurs.
partition groups	1024	(1024)	Number of open partitions during a database load or upgrade – rarely needs changed
partition spinlock ratio	10	(10)	Rarely a problem – watch sp_sysmon for 'Partition Spinlock Contention' and reduce by 2 if it occurs

Configuration Parameter	Default	Recommendation	Tuning Considerations
number of open partitions	500	(see comments)	Both tables and indexes can be partitioned. If using partitioned tables, start with the higher of the number of indexes or the number of rows in syspartitions for most active databases. Monitor with sp_monitorconfig and increase by 10% if >85% active. If not using partitioned tables, simply set to number of open indexes.

In addition to the above, consider using dbcc tune(des_bind) on the following objects if Object Manager Spinlock Contention exceeds 3-5%:

- Sequential key tables (and stored procs used to generate the key values)
- Main transaction tables and transaction history tables as well as any OLTP stored procs, defaults or rules. Reporting procs can be ignored.
- Workflow/job queue tables (and stored procs used to manipulate the queue)

Create a boot script with these in it to facilitate recreating after a reboot of ASE.

Transaction Log & Log Semaphore

One of the biggest impediments to system throughput and scalability is the 'durability' requirement of the ACID. To be durable, the system must be able to recover without any data loss. Unfortunately, this translates either into a requirement that every page write must be flushed to disk when the transaction commits (an impossible task – especially with datarows locking), or the DBMS has to journal changes to a sequential record of modifications. All of the commercial DBMS vendors implement the concept of a journal or transaction log – in fact, most OS's also implement the same idea of a journal to avoid huge data loss for file system devices. Part of the problem with this is that if you reflect back on the hardware discussion, the transaction log must be retained on disk – the slowest part of the computer.

The problem is that most DBA's do not understand the importance of performance on the device and frequently will blame (for instance) the log semaphore for restricting throughput, when in reality the problem is caused by the device itself, the cache configuration or application failing to leverage the different types of logging available. In this section, we will take a look at:

- How transaction logging works and which WaitEvents signal where the problems are.
- What device decisions should be made with respect to the transaction log.
- How to optimize the user log cache and log IO size as well as understanding causes of user log cache flushes to primary transaction log.
- How to exploit explicit transactions via micro-batching to improve throughput.
- The ability to reduce the impact and contention on tempdb by avoiding disk writes and logging through caching in a user log cache dedicated to tempdb.
- The use cases and optimizations available in the Asynchronous Log Service, Delayed Commit and Reduced Durability Databases with respect to reducing the impact of transaction logging.

Transaction Logging in ASE

The ASE transaction log is a typical write-ahead log used by most commercial DBMS's. It is used for a variety of tasks including:

Checkpoint processing – scans the transaction log back to the last checkpoint to determine which data pages have been modified and should be flushed to disk to preserve quick recovery times.

Trigger invocations – scans the transaction log to the beginning of the transaction to build the #inserted and #deleted tables.

Post commit processing – scans the transaction log on large deferred operations to actually perform the deferred inserts, updates or deletes.

Replication Agent – scans the transaction log between markers to generate the LTL to transfer the log records to the Replication Server.

Dump transaction – reads the transaction log back to the last dump/truncation point in order to save the transaction log records to disk and then truncate up to the oldest open transaction.

Rollback of large transactions – transactions that exceed the ULC size or were flushed due to other ULC flushing reasons have the log records in the transaction log. When the transaction needs to be rolled back, these before and after log records are used to 'undo' the data modifications of the transaction to be rolled back.

Recovery from Shutdown/Crash – The well known involvement of rolling forward committed and rolling back uncommitted transactions at the point of a shutdown or server failure.

Transaction Log Space Allocation

The log is represented by the *syslogs* system table in each database that similar to sysobjects is essentially a pointer to the actual log records on disk. It operates similar to a circular list by starting at the top of the log with the lowest logical page id and continuing through to the largest logical page id for the various disk fragment in the *sysusages* table. Once the largest logical page id is reached, the log wraps around and restarts. Consider the following output for a single database from *sysusages*:

dbid	segmap	lstart	size	vstart	pad	unreservedpgs	crdate	vdevno
6	3	0	102400	307200		101180	2009-10-11 21:00:56.983	7
6	4	102400	64000	256000		63757	2009-10-11 21:00:56.983	8
6	3	166400	76800	153600		76500	2009-12-21 09:07:29.436	7
6	3	243200	128000	0		127500	2009-12-21 09:08:00.436	17
6	4	371200	64000	128000		63742	2009-12-21 09:08:20.5	8
6	3	435200	262144	0		261120	2010-07-21 10:27:03.093	25

Since the log segment's segment id is bit '2' or value of 4, any of the above records with a segmap & 4 = 4 would contain log fragments. For the above example, it is those highlighted. As a result, the linear appearance of the transaction log would be similar to:

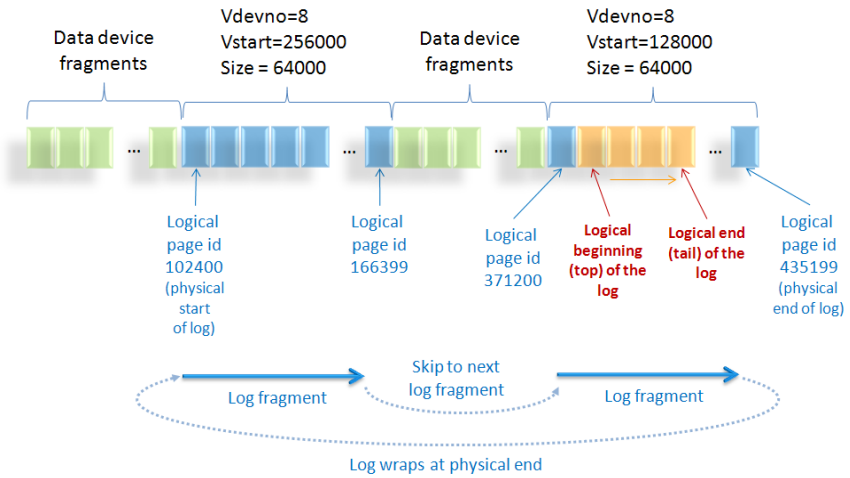


Figure 6 - Logical Page Orientation for Example Log Device Fragments

Note that the logical beginning of the log is at the first allocated (used) page in the transaction log and the end of the log is the latest (or most current) used page in the log. When a log truncation happens, the other log pages are deallocated – once the log expands into the next segment, they are reallocated. Once the physical end of the log device fragments is reached, it simply wraps around to the first logical page id for the first log fragment. Note that in the above – likely due to an allocation to another database that was later dropped, the first log fragment is actually at a higher virtual starting address on the log device (256000) than the second log fragment at vstart=128000. However, the lstart values defines the order in which the device fragments are used as illustrated in the bottom of the above diagram.

In the above, the active part of the log was pictured in the second fragment as just an example. The reason it was done that way was because of the unreserved pages result showing that was where more pages were reserved from the log – but that is not quite an accurate way to find the log boundaries as it often is not up to date. One way to find the active portion of the transaction log is to simply query the syslogshold table and look for the oldest open transaction:

```
1> select * from master..syslogshold
2> go
 dbid   reserved    spid   page        xactid         masterxactid   starttime                  name
xloid
 ------ ----------- ------ ----------- -------------- -------------- -------------------------- ------
------------------------------------------------------------ -----------
     11           0      0        5436 0x000000000000 0x000000000000        Mar 17 2010  4:00AM
$replication_truncation_point                                                    0
     10           0      0        5438 0x000000000000 0x000000000000        Mar 17 2010  4:04AM
$replication_truncation_point                                                    0

(2 rows affected)
```

Note that the page reported is the logical page within each database – so the numbers can be close (as illustrated) or even identical in some cases. One of the issues with syslogshold is that it doesn't show the active log or the end of the log for transaction logs in which there is no currently open transactions. You can force it to do so by simply doing an explicit begin transaction, modifying enough records to overflow the ULC, then looking at syslogshold – then rollback the transaction.

Log Writes, the ULC and Log Cache

The process of writing to the transaction log is a multi-stage process for each SPID. The full process begins with the statement execution and ends (in the simplest form) with the transaction commit.

1. As the SPID processes the statement, if the statement is a DML statement that modifies rows, the SPID starts to identify rows to be modified that it will need to write to its Private Log Cache (PLC) or more commonly known as the User Log Cache (ULC). Before writing to the ULC, the SPID has to grab the spinlock on the spinlock protecting its ULC (sp_configure 'user log cache spinlock ratio')
2. As the rows to be modified are identified, the SPID writes the row images to the ULC. As rows are logged into the ULC, if a modified row image needs to be restored due to 'ignore dupe row' or other index option, the process will add a Compensating Log Record (CLR) that effectively "undoes" the modification needing to be restored. Note, this is not a rollback.
3. Once the ULC is full, a commit is reached or other reason that triggers a ULC flush, the SPID tries to get a lock on the last log page via the log semaphore.
4. Once the log semaphore is granted, since the transaction log is contained in a normal data cache with buffer pools and concurrent accesses, the SPID has to get the spinlock on the cache the log is using.
5. Once it has both the log semaphore and the cache spinlock, the ULC records are copied to the log page in the log cache. As each log page is filled in cache, a new log page is allocated in cache by clearing a page from the LRU end of the cache (irrespective of the object it belongs to) and appending on the MRU.
6. Once all the log records are copied to the cache, the process flushes the cache by issuing physical writes on each log page in sequence starting with the first page written up to the current log page. The last log page may not be written to disk immediately if not full due to the group commit sleep implementation. Once the log pages have been modified in the log cache, the spinlock on the cache containing the log pages is released.
7. Once all the log pages have been written to disk – including the last log page, the transaction is considered committed, the SPID releases the log semaphore and the next statement in the batch or next batch begins processing.

Remember, the actual data changes to *data* or *index* pages is modified in data cache (keeping in mind log pages are written immediately). Those physical writes will be scheduled by the housekeeper or checkpoint processes – or possibly another SPID if a dirty page crosses the wash marker. The below is an illustration of the above process:

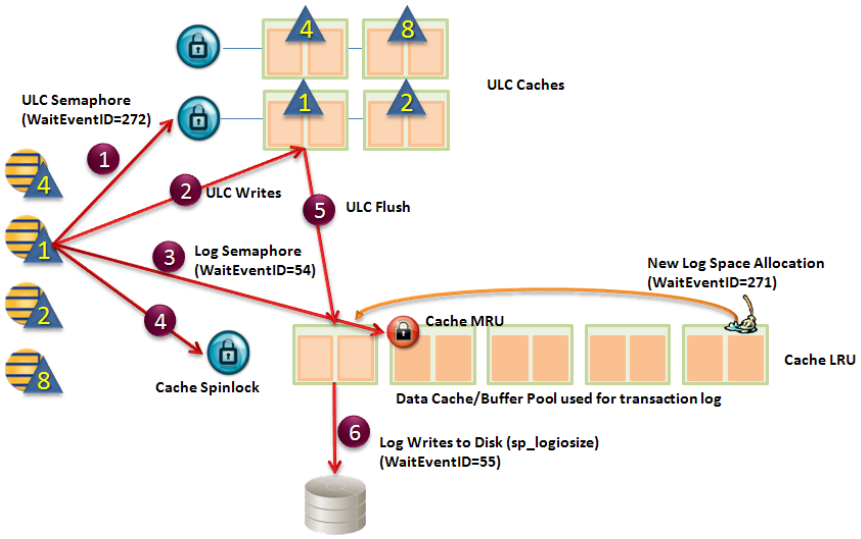


Figure 7 – Normal Transaction Logging Process

There are a few items of interest in the above picture. A common misconception is that the log semaphore is a spinlock. It is not. When attempting to grab a spinlock, a process will continue executing, polling on the spinlock until it becomes available. With a semaphore (aka logical lock), the process is put to sleep until the lock is released. It is much more similar to the normal datapage exclusive lock – in fact, the WaitEvent for lock contention is WaitEventID=150 – "waiting for a lock" in ASE 15, but previously (in 12.5) it was "waiting for a semaphore". The difference is that a datapage lock is controlled via a lock spinlock and lock chain and allows multiple shared locks on the same page. The log semaphore is a logical lock on the last log page. Just like with an exclusive logical lock, if the SPID is able to get the log semaphore, it writes to the log immediately. If, however, the lock is currently held by another SPID, the SPID requesting the lock goes to sleep. Once a process finishes flushing its ULC, there is a new last page of the log, and the log semaphore location is updated. The log semaphore location is tracked in the DBTable structure:

```
1> use demo_db
2> go
1> dbcc traceon(3604)
2> go
DBCC execution completed. If DBCC printed error messages, contact a user with System Administrator
(SA) role.
1> dbcc dbtable('demo_db')
2> go

DBTABLES (ACTIVE):

DB Table at: 0x0000000029C4C8C0


dbt_dbid=8   dbt_stat=0xc (0x0008 (DBT_AUTOTRUNC), 0x0004 (DBT_SELBULK))
dbt_extstat=0x0 (0x0000)
dbt_stat2=0xffff8003 (0x8000 (DBT2_MIXED_LOG_DATA), 0x0002 (DBT2_NOACCOUNT), 0x0001 (DBT2_LOGFULLABT))
dbt_stat3=0x20800 (0x00020000 (DBT3_SYSPARTITIONS_EXISTS), 0x00000800 (DBT3_DELAYED_COMMIT))
dbt_stat4=0x0 (0x00000000)
dbt_runstat=0x0(0x0000)
dbt_state=0x2(0x0002 (DBST_ACTIVE))   dbt_keep=1  dbt_hdeskeep=0 dbt_next=0x0000000029C44C20
dbt_systask_keep=0 dbt_detachxact_keep 0
dbt_lock=0  dbt_dbaid=1
dbt_verstimestamp= Oct 22 2009  6:28AM   dbt_dbname=demo_db
dbt_logrows=0
dbt_lastlogbp=0x00000000438A06F0
dbt_logsema=0000000003BAEE40
dbt_nextseq=12  dbt_oldseq=12

-- after deleting all the rows in a dummy table and inserting 1000 new ones…

1> dbcc dbtable('demo_db')
```

```
DBTABLES (ACTIVE):

DB Table at: 0x0000000029C4C8C0


dbt_dbid=8   dbt_stat=0xc (0x0008 (DBT_AUTOTRUNC), 0x0004 (DBT_SELBULK))
dbt_extstat=0x0 (0x0000)
dbt_stat2=0xffff8003 (0x8000 (DBT2_MIXED_LOG_DATA), 0x0002 (DBT2_NOACCOUNT), 0x0001 (DBT2_LOGFULLABT))
dbt_stat3=0x220800 (0x00200000 (DBT3_WROTE_DELAYED_COMMIT_MSG), 0x00020000
(DBT3_SYSPARTITIONS_EXISTS), 0x00000800 (DBT3_DELAYED_COMMIT))
dbt_stat4=0x0 (0x00000000)
dbt_runstat=0x0(0x0000)
dbt_state=0x2(0x0002 (DBST_ACTIVE))   dbt_keep=2  dbt_hdeskeep=0 dbt_next=0x0000000029C44C20
dbt_systask_keep=0 dbt_detachxact_keep 0
dbt_lock=0  dbt_dbaid=1
dbt_verstimestamp= Oct 22 2009  6:28AM   dbt_dbname=demo_db
dbt_logrows=9
dbt_lastlogbp=0x0000000004389DF10
dbt_logsema=0000000003BCEE40
dbt_nextseq=5653  dbt_oldseq=5638
```

Note that in the above example, the log semaphore location changed as did the log page timestamp sequence (dbt_nextseq).

From an MDA perspective, there a several ULC or Log writing WaitEvents that can happen – although the ULC WaitEvents occur very rarely as rarely does a process have to wait on those events. Consider the following list and their descriptions from monWaitEventInfo:

WaitEventID	Description	Comments
54	waiting for write of the last log page to complete	Waiting for log semaphore
55	wait for i/o to finish after writing last log page	Waiting for log writes to disk
89	wait for space in ULC to preallocate CLR	(rare)
90	wait after error in ULC while logging CLR	(rare)
122	wait for space in ULC to log commit record	(rare)
123	wait after error logging commit record in ULC	(rare)
140	wait to grab spinlock	Possibly a cache spinlock – note that this is rare as a spinlock is a true spinlock and process doesn't sleep. You will only see this event if a process was forced to sleep (timeslice) while still trying to get a spinlock.
259	waiting until last chance threshold is cleared	Typical log suspend issue due to log full
271	waiting for log space during ULC flush	(rare) likely cause for this was that there was a cache stall in the data cache buffer pool containing the log. Coupled with WaitEventID=140 (above), this should help explain what a dedicated log cache is important for any high volume OLTP database.
272	waiting for lock on ULC	Mostly seen in high volume OLTP systems with a lot of write activity – usually a small value. Can be tuned via sp_configure 'user log cache spinlock ratio' if it becomes a problem

The difference between WaitEventID 54 & 55 is hard to detect from just the description. However, remember, that the only reason why you can't get the log semaphore is if someone else currently has it – and is writing to the last page (ULC flush) or is waiting for the log physical write to complete (in the case a group commit sleep is not used). So the notion of "waiting for the write of the last log page to complete" is simply saying that the SPID is waiting on another SPID to finish its ULC flush to the last log page in cache. WaitEventID=55 is a bit clearer as it states that it is waiting on a physical IO to return on the last log page.

Log Writes & Group Commit Sleeps

A common misunderstanding is that every flush of the ULC to the primary transaction log results in a log write to disk. Early in ASE version 11.0, Sybase attempted to reduce the impact of the log device on throughput by implementing 'group commit sleeps'. With this facility, concurrent users were put to sleep after flushing the ULC to the log cache while another user's transactions were appended to the transaction log. The rationale was that by doing so, it would reduce the number of log writes per log page and thereby improve throughput by reducing the wait time on the log flush to disk. Initially, this was configurable, but today, it is automatic based on the log IO size (sp_logiosize). As a result, the log cache is only written to disk when:

- A process fills the current last log page in the sp_logiosize buffer (e.g. the second page in a 4K buffer pool for a 2K server).
- A process reaches the head of the run queue while sleeping on the log write and another process has not yet forced the log page to write to disk.

The best way to think of it is to pretend you have a system

- that is doing all atomic transactions
- you can fit 2 log records on each log page.
- you have 4 processes each doing the same atomic transactions but in sequence and they are the only 4 processes in the system.
- you have a 4K page server but with an 8K sp_logiosize

What would happen is the following:

- SPID #1 appends its log records at start of first log page – sleeps on WaitEventID=55
- SPID #2 appends its log records. Although this fills a log page, because sp_logiosize is two pages, nothing happens (group commit sleep). SPID #2 sleeps on WaitEventID=55
- SPID #3 appends its log records at start of second log page – sleeps on WaitEventID=55
- SPID #4 appends its log records – fills the sp_logiosize, and log is flushed to disk. Sleeps on WaitEventID=55
- Write to disk completes – all four SPIDS are woken up and control is returned to the clients.

Now, let's make a slight change. Let's say that instead of 2 log records per page, we can get 10 log records per page. Assuming the same sequence we would have the following:

- SPID #1 appends its log records at start of first log page – sleeps on WaitEventID=55
- SPID #2 appends its log records – still on first page of log, sleeps on WaitEventID=55
- SPID #3 appends its log records – still on first page of log, sleeps on WaitEventID=55

- SPID #4 appends its log records – still on first page of log, sleeps on WaitEventID=55
- SPID #1 appears at top of run queue – log is flushed to disk even though only 40% full
- Write to disk completes – all four SPIDS are woken up and control is returned to the clients.
- SPID #1 appends its log records to first log page since it is not full yet – sleeps on WaitEvent=55
- (process continues)

In one sense, if a single SPID is doing a lot of atomic inserts, the end result is that every atomic insert becomes a physical log write if no other process on the system is performing any activity. But that is the only acceptable implementation. If the SPID was allowed to commit and a second insert allocated on the log page without waiting for the flush to disk, a system crash between the two would cause the first insert to be lost as the log page had not yet been written (see delayed commit description later). As you can also see, however, there also is a relationship between the sp_logiosize and transaction sizes and user concurrency. At low concurrency and small transactions, increasing the log IO size will not be effective as the user will simply force a log write before the log page is even full.

Group commit sleeps (along with log semaphore contention) is reported in sp_sysmon via the sections:

```
  Task Context Switches Due To:
    Voluntary Yields               478.3            0.2      867230       1.8 %
    Cache Search Misses           1395.9            0.5     2530709       5.4 %
    System Disk Writes              13.2            0.0       23996       0.1 %
    I/O Pacing                     489.5            0.2      887417       1.9 %
    Logical Lock Contention         11.3            0.0       20434       0.0 %
    Address Lock Contention          0.1            0.0         112       0.0 %
    Latch Contention                 2.3            0.0        4176       0.0 %
    Log Semaphore Contention        11.6            0.0       20977       0.0 %
    PLC Lock Contention              1.9            0.0        3503       0.0 %
    Group Commit Sleeps            126.1            0.0      228583       0.5 %
    Last Log Page Writes           257.5            0.1      466862       1.0 %
    Modify Conflicts                28.9            0.0       52327       0.1 %
    I/O Device Contention            0.0            0.0           0       0.0 %
    Network Packet Received       7050.7            2.5    12782963      27.1 %
    Network Packet Sent           5217.9            1.8     9459981      20.1 %
    Other Causes                 10886.6            3.8    19737421      41.9 %

…

Transaction Profile
-------------------

  Transaction Summary             per sec      per xact       count  % of total
  --------------------------  ------------  ------------  ----------  ----------
    Committed Xacts                2854.9           n/a     5175863         n/a

…
    Total Rows Affected           24695.9           8.7    44773665
…

Transaction Management
----------------------

  ULC Flushes to Xact Log         per sec      per xact       count  % of total
  --------------------------  ------------  ------------  ----------  ----------
    by Full ULC                     80.2            0.0      145370       1.7 %
    by End Transaction            1611.4            0.6     2921460      33.9 %
    by Change of Database           61.5            0.0      111539       1.3 %
    by Single Log Record          1567.5            0.5     2841948      33.0 %
    by Unpin                        49.6            0.0       89846       1.0 %
    by Other                      1386.8            0.5     2514229      29.2 %
  --------------------------  ------------  ------------  ----------
  Total ULC Flushes               4757.0            1.7     8624392
```

```
ULC Log Records                 22288.1           7.8    40408318       n/a
Max ULC Size During Sample          n/a           n/a           0       n/a

ULC Semaphore Requests
  Granted                       52301.4          18.3    94822415     100.0 %
  Waited                            1.9           0.0        3503       0.0 %
-------------------------  ------------  ------------  ----------
Total ULC Semaphore Req         52303.3          18.3    94825918

Log Semaphore Requests
  Granted                        4203.1           1.5     7620299      99.7 %
  Waited                           11.6           0.0       20977       0.3 %
-------------------------  ------------  ------------  ----------
Total Log Semaphore Req          4214.7           1.5     7641276

Transaction Log Writes           1020.0           0.4     1849241       n/a
Transaction Log Alloc            1518.9           0.5     2753780       n/a
Avg # Writes per Log Page            n/a          n/a     0.67153       n/a

Tuning Recommendations for Transaction Management
-------------------------------------------------
- Consider decreasing the 'user log cache size'
  configuration parameter if it is greater than the
  logical database page size.
```

The above example shows ~5 million transactions but only ~1.8 million log writes. We can also infer that because of only 228583 group commit sleeps in those 1.8 million log writes, that most of the transactions were fairly big – but some were smaller than the log IO size. Net result is that the actual number or impact of group commit sleeps was fairly small – probably because a particular user (or users) were doing small transactions was forcing the log to flush. Some of the 5 million transactions could be queries with no DML, but if not, at the most it means that each log page is contains 2.8 transactions. However, the ~5 million transactions only resulted in about 2.75 million space allocations of 1 or more log records on log pages, which when compared to the 1.85 million log writes results in the average number of writes per log page being 0.67.

In high volume systems, the time spent sleeping in a group commit sleep stage is very small compared to the time spent sleeping waiting for the log page to be written to disk. However, if log semaphore contention is very high, then the current process may end up waiting longer for the next process to get through the contention on the log semaphore to fill the page. The point being made is that if the log semaphore contention is very high, it may make the log device appear slower than it is – although normally you can think of WaitEventID=55 as being predominantly the log device speed.

Reading the Transaction Log

The log records can be read by using the dbcc log command:

```
-- conical form
-- dbcc log(<dbid>,<objid>,<pageid>,<rowid>,<num records>,<category>,<print option>,<indid>,<ptnid>)

dump the last two transactions for spid 5 in dbid=5

1> dbcc log(5,-5, -3, 0, -2, 0, 1)
2> go

LOG RECORDS:
        BEGINXACT       (1793, 2)
        attcnt=1 rno=2 op=0 padlen=0 xactid=(1793, 2) len=60 status=0x0000
        masterid=(0, 0)         lastrec=(0, 0)
        xstat=XBEG_ENDXACT,
        spid=5 suid=3 uid=3 masterdbid=0 mastersite=0 endstat=3
        name=ins   time=Aug 22 1994 10:57AM

        BEGINXACT       (1807, 20)
        attcnt=1 rno=20 op=0 padlen=0 xactid=(1807, 20) len=60 status=0x0005
        masterid=(0, 0)         lastrec=(0, 0)
```

```
        xstat=XBEG_ENDXACT,
        spid=5 suid=3 uid=3 masterdbid=0 mastersite=0 endstat=3
        name=tla   time=Aug 22 1994 10:57AM

DBCC execution completed. If DBCC printed error messages, contact a user with
System Administrator (SA) role.
```

Dump out the entire transaction for transaction ID (1807, 20) (from above) in dbid=5

```
1>dbcc log(5, 1, 1807, 20, 0, -1, 1)
2> go

LOG RECORDS:
        BEGINXACT       (1807, 20)
        attcnt=1 rno=20 op=0 padlen=0 xactid=(1807, 20) len=60 status=0x0005
        masterid=(0, 0)         lastrec=(0, 0)
        xstat=XBEG_ENDXACT,
        spid=5 suid=3 uid=3 masterdbid=0 mastersite=0 endstat=3
        name=tla   time=Aug 22 1994 10:57AM

        DBNEXTID        (1807, 21)
        attcnt=1 rno=21 op=14 padlen=0 xactid=(1807, 20) len=24 status=0xffffbdf
8
        dbid=5 nextid=992006565

        INSERT  (1807, 22)
        attcnt=1 rno=22 op=4 padlen=1 xactid=(1807, 20) len=76 status=0x0000
        oampg=51 pageno=54 offset=352 status=0x00
        old ts=0x0001 0x00003400   new ts=0x0001 0x00003404

        INSERT  (1807, 23)
        attcnt=1 rno=23 op=4 padlen=3 xactid=(1807, 20) len=124 status=0x0000
        oampg=2 pageno=3 offset=422 status=0x00
        old ts=0x0001 0x00003402   new ts=0x0001 0x00003405

        IINSERT (1807, 24)
        attcnt=1 rno=24 op=7 padlen=3 xactid=(1807, 20) len=64 status=0x0000
        oampg=2 pageno=16 offset=647 status=0x00
        old ts=0x0001 0x00003401   new ts=0x0001 0x00003406

        OAMCREATE       (1807, 25)
        attcnt=1 rno=25 op=41 padlen=0 xactid=(1807, 20) len=84 status=0x0000
        Pagehdr: pageno=856 nextpg=856 prevpg=856
                 ts=0x0001 0x00003407 stat=32768, objid= 976006508, oampg= 0
        old next ts=0x0000 0x00000000 new next ts=0x0000 0x00000000
        old prev ts=0x0000 0x00000000 new prev ts=0x0000 0x00000000

        OAMINSERT       (1807, 26)
        attcnt=1 rno=26 op=39 padlen=0 xactid=(1807, 20) len=60 status=0x0000
        oampg= 856, allocpg= 768, xinspg= 856
        prev ts=0x0001 0x00003407 new ts=0x0001 0x00003409
        hdr prev ts=0x0001 0x00003407 hdr new ts=0x0001 0x00003409

        ALLOC   (1807, 27)
        attcnt=1 rno=27 op=13 padlen=0 xactid=(1807, 20) len=84 status=0x0000
        Pagehdr: pageno=857 nextpg=0 prevpg=0
                 ts=0x0001 0x0000340a stat=1, objid= 976006508, oampg= 856
        old next ts=0x0018 0x00000000 new next ts=0x0054 0x0000003c
        old prev ts=0x0000 0x00000005 new prev ts=0x0054 0x00000000

        INSERT  (1793, 0)
        attcnt=1 rno=0 op=4 padlen=0 xactid=(1807, 20) len=112 status=0x0000
        oampg=26 pageno=27 offset=1716 status=0x00
        old ts=0x0001 0x00003403   new ts=0x0001 0x0000340c

        ENDXACT (1793, 1)
        attcnt=1 rno=1 op=30 padlen=0 xactid=(1807, 20) len=28 status=0x0000
        endstat=COMMIT time=Aug 22 1994 10:57AM

DBCC execution completed. If DBCC printed error messages, contact a user with
System Administrator (SA) role.
```

Dump the last checkpoint from the log for dbid=5 (Checkpoint is category 17)

```
1> dbcc log(5,0, 0, 0, -1, 17, 0)
2> go

LOG RECORDS:
        CHECKPOINT      (1814, 6)
        attcnt=1 rno=6 op=17 padlen=2 xactid=(0, 0) len=64 status=0x0000
        rows=0, pages=0 extents=0
        timestamp=0x0001 0x000032b5  xstat=0x0000  no active xacts
```

Dump the last 50 records from the log

```
1> dbcc log(11,0, 0, 0, -50, -1, 1)
2> go
DBCC execution completed. If DBCC printed error messages, contact a user with System Administrator
(SA) role.
LOG SCAN DEFINITION:
        Database id : 11
        Backward scan: starting at end of log
        maximum of 50 log records.
LOG RECORDS:
        ENDXACT                  (5478,26)       sessionid=5477,8
        attcnt=1 rno=26 op=30 padlen=0 sessionid=5477,8 len=28
        odc_stat=0x0000 (0x0000)
        loh_status: 0x0 (0x00000000)
        endstat=COMMIT time=Jul 18 2007 11:26:40:106PM
        xstat=0x0 []
        INSERT                   (5478,25)       sessionid=5477,8 [REPLICATE]
        attcnt=1 rno=25 op=4 padlen=1 sessionid=5477,8 len=140
        odc_stat=0x0008 (0x0008 (LHSC_REPLICATE))
        loh_status: 0x8 (0x00000008 (LHSC_REPLICATE))
        objid=576002052  ptnid=576002052 pageno=968 offset=1893 status=0x814 (0x0800 (XSTAT_EXPAGE),
0x0010 (XSTAT_REV_UPD_SCAN), 0x0004 (XSTAT_XUPDATE))
        cid=0 indid=0
        old ts=0x0000 0x00003726  new ts=0x0000 0x00003727
        DELETE                   (5478,24)       sessionid=5477,8 [REPLICATE]
        attcnt=1 rno=24 op=5 padlen=1 sessionid=5477,8 len=140
        odc_stat=0x0008 (0x0008 (LHSC_REPLICATE))
        loh_status: 0x8 (0x00000008 (LHSC_REPLICATE))
        objid=576002052  ptnid=576002052 pageno=968 offset=1893 status=0x814 (0x0800 (XSTAT_EXPAGE),

…(output clipped to condense)…

INSERT                    (5478,21)       sessionid=5477,8 [REPLICATE]
        attcnt=1 rno=21 op=4 padlen=3 sessionid=5477,8 len=136
        odc_stat=0x0008 (0x0008 (LHSC_REPLICATE))
        loh_status: 0x8 (0x00000008 (LHSC_REPLICATE))
        objid=576002052  ptnid=576002052 pageno=968 offset=1727 status=0x814 (0x0800 (XSTAT_EXPAGE),
0x0010 (XSTAT_REV_UPD_SCAN), 0x0004 (XSTAT_XUPDATE))
        cid=0 indid=0
        old ts=0x0000 0x00003722  new ts=0x0000 0x00003723
        DELETE                   (5478,20)       sessionid=5477,8 [REPLICATE]
        attcnt=1 rno=20 op=5 padlen=3 sessionid=5477,8 len=136
        odc_stat=0x0008 (0x0008 (LHSC_REPLICATE))
        loh_status: 0x8 (0x00000008 (LHSC_REPLICATE))
        objid=576002052  ptnid=576002052 pageno=968 offset=1727 status=0x814 (0x0800 (XSTAT_EXPAGE),
0x0010 (XSTAT_REV_UPD_SCAN), 0x0004 (XSTAT_XUPDATE))
        cid=0 indid=0
        old ts=0x0000 0x00003721  new ts=0x0000 0x00003722
        …
        BEGINXACT                (5477,8)        sessionid=5477,8
        attcnt=1 rno=8 op=0 padlen=2 sessionid=5477,8 len=132
        odc_stat=0x0000 (0x0000)
        loh_status: 0x0 (0x00000000)
        masterxsid=(invalid sessionid)
        xstat=XBEG_ENDXACT,XBEG_USERINFO_EXT,
        spid=20 suid=1 uid=1 masterdbid=0 dtmcord=0
        name=$upd   time=Jul 18 2007 11:26:40:106PM
        xversion=0 xextension.encrypt_method=2 xextension.encrypt_area_len=62
        xextension.encrypt_area:
2B5E4438 (     0):  bfbfccad e792fede efd7f7c5 f5c5f2d2  ................
2B5E4448 (    16):  e3d2e8da ecbcf1bb cea282b3 8bab99a9  ................
2B5E4458 (    32):  99ae8ebf 8eb486b0 e0ade792 fedeefd7  ................
2B5E4468 (    48):  f7c5f5c5 f2d2e3d2 e8daecbc f1bb      ..............
        CHECKPOINT               (5477,7)        sessionid=5477,7
        attcnt=1 rno=7 op=17 padlen=0 sessionid=5477,7 len=60
        odc_stat=0x0008 (0x0008 (LHSC_REPLICATE))
        loh_status: 0x8 (0x00000008 (LHSC_REPLICATE))
        rows=0, pages=0 extents=0 logvers=7
        timestamp=0x0000 0x000036e8  xstat=0x0000 (0x0000)
        time=Jul 12 2007  2:10PM
        no active xacts
```

The last example illustrates when replication is enabled and the appropriate replication bit is flagged on the individual log records.

Log Semaphore Contention

In order to guarantee that after recovery, the database is consistent when it shutdown, the log records are written in serialized order. The serialization for ASE transaction logs is enforced

through the use the log semaphore as described earlier. Multiple processes waiting on the log semaphore is often referred to as 'log semaphore contention'. Unfortunately, most DBA's simply associate the log semaphore contention with the semaphore and blame it for the contention vs. the underlying causes. The causes for log semaphore contention can generally be due to one of the following reasons:

- Slow physical devices for the transaction log
- Small ULC cache size (and log write inefficiency)
- High atomic DML activity due to lack of exploiting transaction controls.
- High atomic DML activity due to business/architecture requirements.
- High concurrency in tempdb (user defined or system).
- High concurrency on large/very large SMP platforms.

Each of these have different optimal solutions – using a different solution may not only make the problem worse, but also mask the symptoms due to normal monitoring not seeing the symptoms hidden by the wrong solution. Also, note that there is a slight difference between log semaphore contention and log waits. As noted earlier, log semaphore contention can be measured at a high level via sp_sysmon or at a much more discrete level via the MDA wait event WaitEventID=54 – 'Waiting for the write of the last log page to complete'. However, some changes in logging options can reduce this – yet the process may still be waiting to append to the log – which is still a problem even if the log semaphore is no longer causing it. The best way to measure log contention – whether or not caused by the log semaphore – is to use monOpenDatabases:

```
create existing table monOpenDatabases (
        DBID                            int,
        InstanceID                      tinyint,
        BackupInProgress                int,
        LastBackupFailed                int,
        TransactionLogFull              int,
        SuspendedProcesses              int,
        AppendLogRequests               int,
        AppendLogWaits                  int,
        DBName                          varchar(30) NULL,
        BackupStartTime                 datetime NULL,
        LastCheckpointTime              datetime NULL,
        LastTranLogDumpTime             datetime NULL,
        QuiesceTag                      varchar(30) NULL,
)
external procedure
at "@SERVER@...$monOpenDatabases"
go
```

Contention is simply the percentage of AppendLogWaits vs. AppendLogRequests – and ideally should be less than 5-10% on a steady basis during normal processing. Both of these values are tracked in the DBTable structure and simply records the number of times the transaction log was appended to, and the number of times that attempts to append to the log were forced to wait – whatever the cause. In the later discussion on ALS, one of the other AppendLogWait causes other than the log semaphore will be discussed. Another is quite simply a full transaction log that results in a log suspend.

This information is best plotted over time vs. simply aggregated – and is much more accurate than short sp_sysmon snapshots taken once per hour. For example, during an entire business day, a particular customer's key transaction database showed the following characteristics:

DBID	DBName	LogRequests	LogWaits	LogContention

DBID	DBName	LogRequests	LogWaits	LogContention
22	Database_1	6,111,526	899,828	6.6
27	dbccdb	0	0	1.6
6	Database_2	692,254	3,826	0.4
17	Database_3	213,412	628	0.2
46	Usertempdb_7	586,879	369	0
2	tempdb	773,810	343	0

The average log contention of 6.6% doesn't look bad until you plot it over time using monOpenDatabases samples

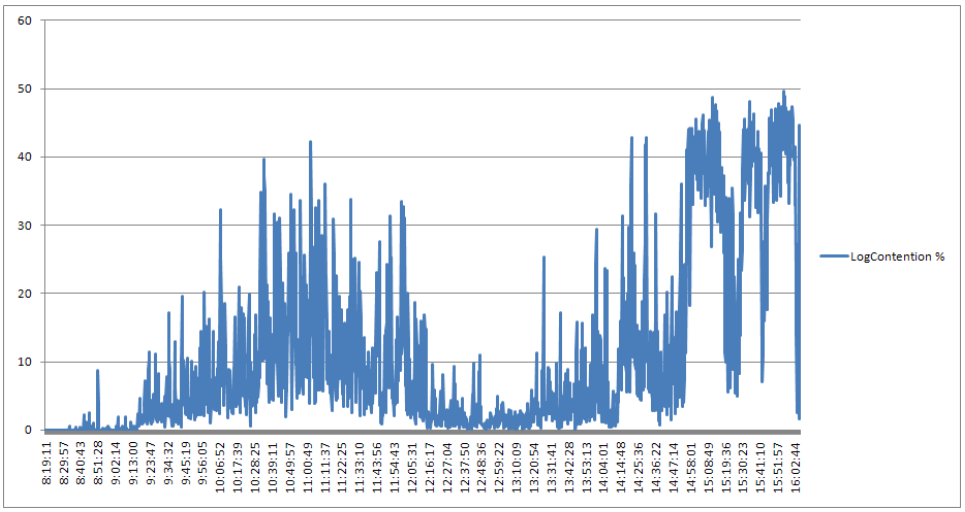


Figure 8 – Log Contention % Plotted Over Time from monOpenDatabases

Obviously the log contention is quite severe and is impacting performance during key peak business times.

Once we understand that log semaphore contention remains big factor in the system throughput, the next thing is to reduce the log semaphore contention as much as possible – by considering other options:

- Split the database into multiple logical databases
- Optimize the log device
- Tune the ULC and log IO size
- Exploit the ULC to minimize the number of writes per log page
- Increase the session tempdb log cache size for tempdb to prevent log writes
- Use the Asynchronous Log Service
- Use delayed commit
- Use a Reduced Durability Database

The first one is a design issue and often requires application changes – although its effectiveness is indisputable if the partitioning of the data between the different physical databases is done along transaction boundaries. This takes considerable work, however, and can require careful analysis of which tables are impacted by which transactions – or which

transactions impact different subsets of data in the same tables. Additionally, the impact extends to which reports or queries will now be forced to union data from the different databases. As a result, while it is quite effective for OLTP applications, it can require trade-offs for reporting applications. Additionally, it often is an approach that only works when the application can be complete retooled as part of an entire technology stack refresh – and not just a tuning exercise or technology update. The other techniques will be discussed the following paragraphs.

Transaction Log Physical Devices

Often we can't do anything about the log device – we are stuck with the SAN configuration that the business was willing to pay for (e.g. tier 2 storage with RAID 6). Even if that is true, understanding that the problem is the log device helps determine what is possible and what isn't – especially in reducing time wasted on initiatives that wouldn't help such as further tuning the ULC. But a careful look at the log device and the types of physical IOs that are happening against it is certainly a key consideration.

Ideally, the transaction log is written to but never read – except from a log cache. In any case, since the log semaphore is held until the physical write is completed, a slow log device not only decreases throughput for the entire system, it also increase the log semaphore contention. As a result, the transaction log device should be one of the fastest on the system. This is a bit interesting – as many immediately assume this means using RAID 1+0 vs. RAID 5 for the log device, and generally, that is a good rule for high performance systems. However, the actual answer is dependent on the storage vendor and the transaction rates involved. For enterprise SAN storage, almost all of the penalty for RAID 5 or RAID 6 over RAID 0 or RAID 10 is during physical read operations – which unless the log cache is not big enough, should not be happening. Due to the speed of RAID chipsets and buffering in RAID cache or the SAN cache, the overhead of parity calculations as part of the writing process is largely negligible compared to simplistic disk striping. However, for RAID controller cards in the host system or if the SAN write cache is likely fully utilized, the RAID 5 or 6 parity calculation can result in considerable overhead – which taken in conjunction with fewer devices used for striping (e.g. a 5 disk raidset will stripe data on 4 and use the 5th for parity for RAID 5 whereas RAID 0 will stripe across all 5), could very negatively impact system throughput.

Remember that each log page write is an IO operation. The faster the overall device speed is with respect to the number of IOPS sustainable, the faster and higher the throughput of the system will be perceived. Consider the following table that illustrates the number of IOPS and relative throughput for a system if the transaction log device were on such an implementation:

Device Type	Typical IOPS	Typical ms/IO	Xactn/Sec
RAID 6	600	10-25	40-100
RAID 5	800	2-6	166-500
RAID 0 (RAID 10)	1000	1-2	500-1000
FC SCSI (tier 1)	200-400	2-6	166-500
SAS SCSI (tier 2)	150-200	4-8	125-250
SATA (tier 3)	100-150	6-10	100-166
SATA/SAS SSD	10000-20000	0.05-0.1	10,000-20,000
PCIe SSD	100000-200000	0.01-0.005	100,000-200,000

Obviously from the above chart, the best device to use for the transaction log if log write speed is causing semaphore contention is to use an SSD.

The quickest way to determine the log device speed in any system is to look at monDeviceIO or monIOQueue. Consider the following analysis from monDeviceIO from a customer system:

LogicalName	Reads	APFReads	APF_Pct	Writes	TotalIOs	Write_Pct	IOTime_ms	msPerIO
dev27dat	2,228,537	150,831	6	476,731	2,705,268	17	17,776,500	6
dev08log	6,531	0	0	2,553,000	2,559,531	99	5,854,900	2
dev19log	5,114	42	0	1,798,760	1,803,874	99	3,519,400	1
dev16tmp	1,573	0	0	1,627,347	1,628,920	99	3,627,500	2
dev54dat	641,019	33,914	5	609,677	1,250,696	48	23,811,700	19
dev01web10	233,859	18,434	7	415,298	649,157	63	11,350,700	17
dev02tmp	1,453	0	0	642,135	643,588	99	1,889,800	2
dev07dat	334,844	34,194	10	305,257	640,101	47	4,626,300	7
dev48tmp	13	0	0	528,948	528,961	99	9,894,700	18
dev49tmp	694	0	0	493,998	494,692	99	8,031,600	16
dev60tmp	1,216	0	0	477,308	478,524	99	3,574,600	7
dev61tmp	1,336	0	0	426,461	427,797	99	9,918,000	23

Both highlighted devices are showing ~2ms/IO (dev19log actual average is 1.9 before truncation by the datatypes cast) which is on the slow side as it limits throughput to ~500 transactions/second if atomic transactions.

Another way to determine if the log write flush/log device is the cause of the contention is to look at monSysWaits or monProcessWaits and specifically the relationship between WaitEventID=55 and other waits. Consider the following output from monSysWaits from ASE 12.5

WaitEventID	Waits	WaitTime	Description
29	1,399,734,780	4,929,606	wait for buffer read to complete
215	1,127,548,743	27,844,254	waiting on run queue after sleep
35	670,307,183	4,153,181	wait for buffer validation to complete
179	335,988,317	10,035,324	waiting while no network read or write is required
250	324,655,266	167,672,101	waiting for incoming network data
124	256,994,610	6,105,113	wait for someone else to finish reading in mass
209	75,463,669	340,471	waiting for a pipe buffer to read
251	62,546,271	344,880	waiting for network send to complete
41	58,470,129	3,473,384	wait to acquire latch
31	32,361,806	36,401	wait for buffer write to complete
214	19,911,597	1,403,956	waiting on run queue after yield
150	18,083,160	23,776,944	waiting for semaphore
52	11,842,516	48,193	waiting for disk write to complete
51	9,703,708	27,945	waiting for disk write to complete
55	8,071,811	5,609	waiting for disk write to complete
36	4,774,219	20,192	wait for mass to stop changing
272	3,886,481	134,998	waiting for lock on PLC
54	2,438,135	30,805	waiting for disk write to complete

In this example, the log flush to disk (WaitEventID=55) is extremely fast – with <1ms per IO since ~8 million writes took only 5.6 seconds to achieve. The log semaphore contention (WaitEventID=54) was also very fast at well under 1ms per wait although 600% more than the wait time for the log writes. This would not be uncommon as a SPID sleeps for the log semaphore access if it can't grab it within the current timeslice. Note that the log contention

appears to be about 30% as the ratio of log semaphore waits vs. log flushes to disk (2.4M:8.0M=0.3). Regardless the conclusion from the above is that the log device is fast enough and certainly not a major contributor to system performance concerns as it ranks far below lock contention (WaitEventID=150) and physical IO.

On the other hand consider the following benchmark test scenario in which the monProcessWaits events were aggregated across the individual users in the test scenario (to filter out non-test users such as the monitoring process, system tasks and other users) (focus on the 6th and 7th bars in the graph):

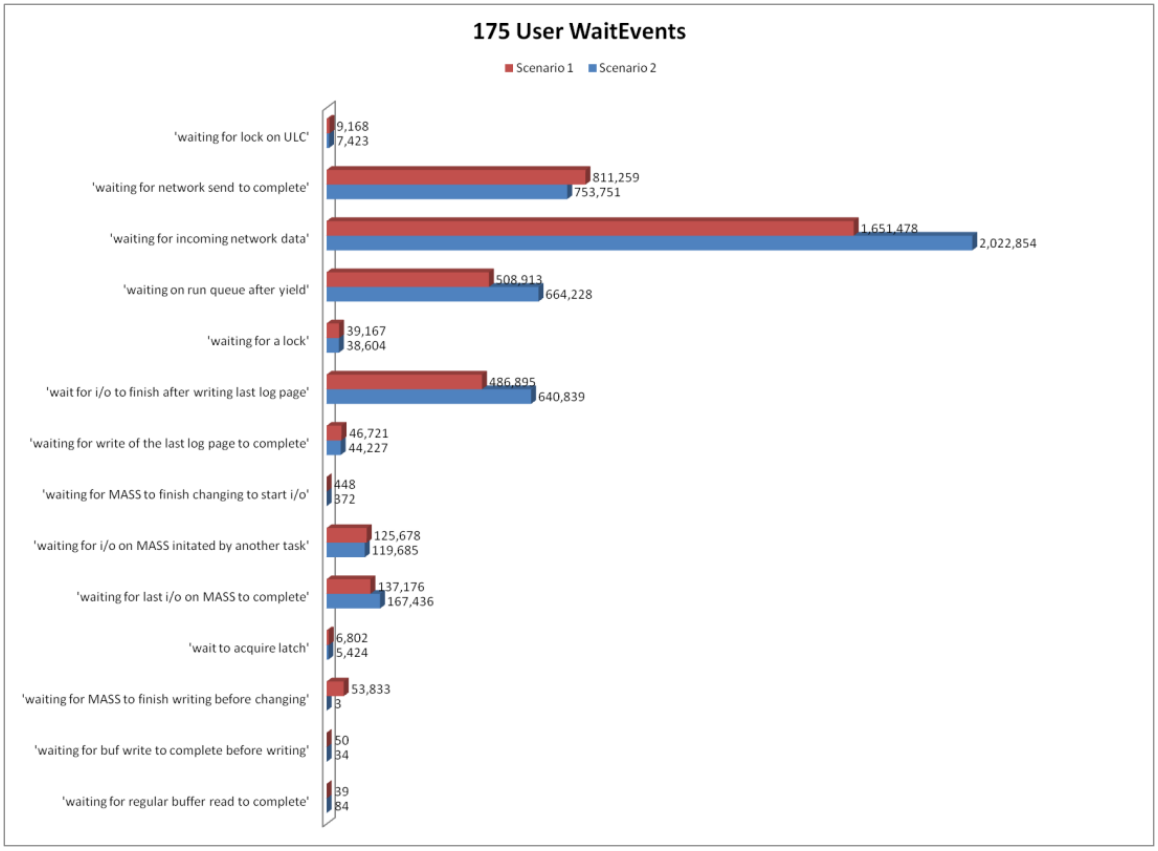


Figure 9 - Aggregated monProcessWaits Waits & Log Semaphore Contention

The log flush writes ('wait for i/o to finish *after* writing last log page' – WaitEventID=55) is extremely high in both test scenarios at ~500K and ~640K waits. The log semaphore waits ('waiting for write of the last log page to complete' – WaitEventID=54) is actually fairly small in comparison - ~45K waits in each case. However, if you graph the wait time, you see something interesting:

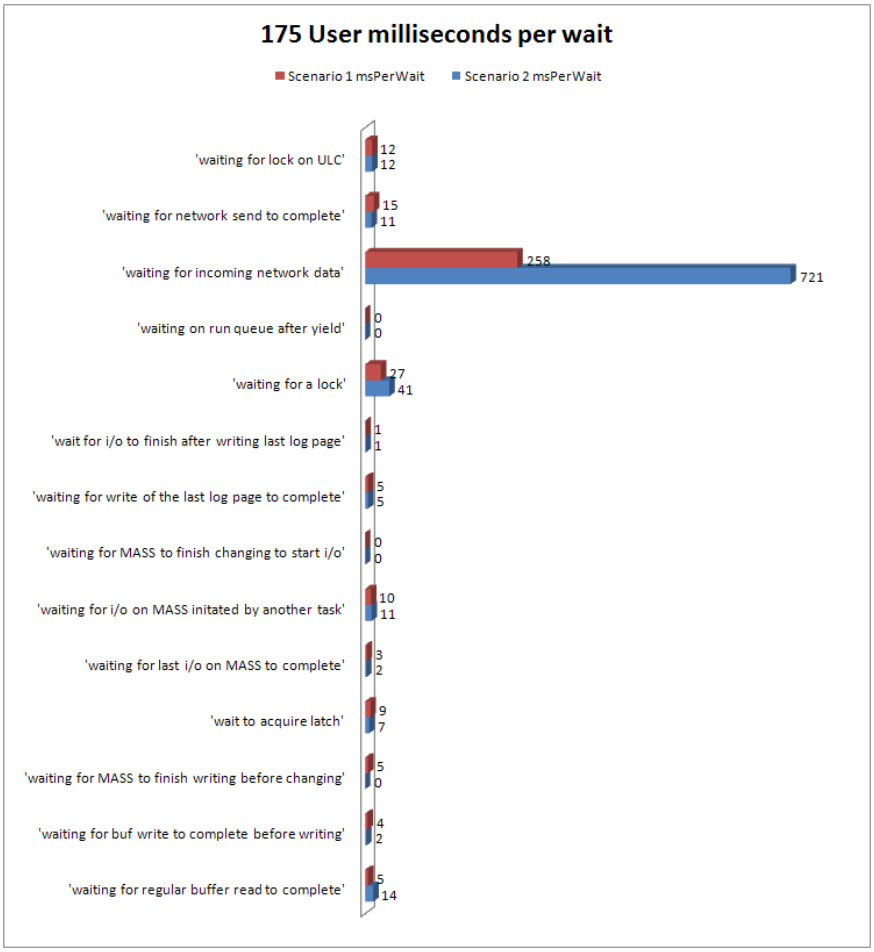


Figure 10 - Aggregated monProcessWaits WaitTime & Log Semaphore Contention

This shows that the device speed is appears quite fast as well at ~1ms per IO while the log semaphore contention was a little slower at ~5ms per attempt. However, consider the raw data as aggregated into a table for scenario #2:

ID	Description	SPIDS	Waits	Wait Time	ms/Wait
29	'waiting for regular buffer read to complete'	47	84	1,200	14
31	'waiting for buf write to complete before writing'	348	34	100	2
36	'waiting for MASS to finish writing before changing'	3	3	0	0
41	'wait to acquire latch'	234	5,424	38,000	7
51	'waiting for last i/o on MASS to complete'	336	167,436	437,300	2
52	'waiting for i/o on MASS initated by another task'	297	119,685	1,361,300	11
53	'waiting for MASS to finish changing to start i/o'	143	372	0	0
54	'waiting for write of the last log page to complete'	296	44,227	239,500	5
55	'wait for i/o to finish after writing last log page'	345	640,839	882,400	1
150	'waiting for a lock'	281	38,604	1,615,800	41
214	'waiting on run queue after yield'	345	664,228	202,900	0
250	'waiting for incoming network data'	348	2,022,854	1,458,992,800	721
251	'waiting for network send to complete'	344	753,751	8,350,300	11
272	'waiting for lock on ULC'	247	7,423	89,700	12

Using the waits for 54 vs. 55, we can see the log semaphore contention was about ~7%. The actual device speed is 1.4ms/IO and although this is appears low on average any attempt to increase throughput would benefit more from speeding up the log device vs. decreasing log semaphore contention.

User Log Cache & Log IO Size

One way to ensure the log write speed is optimal without changing the hardware is to ensure that the ULC size is at least the same size as the database log IO size, the database log IO size is set appropriately – and that the database transaction log is bound to a log cache with the same size buffer as the log IO size. This may sound like an old topic to a lot of DBA's, but the reality is that few systems have this tuned correctly. Most DBA's are aware that the default log IO size is 2 times the server page size – so for a 2K server, this equates to a default log IO size of 4K. However, what they may not be aware of is:

- This default is dependent upon a buffer pool of the same size being available
- If the database was created prior to the buffer pool being available, the log IO size defaults to the server page size and needs to be reset to achieve the stated default of 2x the page size
- The default 'user log cache size' is the same as the server page size – for a 2K server, this means 2048 bytes. Unless this matches the log IO size, the log will not use the larger IO size for log writes.
- This concerns tempdb as well (even more so as described in the next section).

Adding to this problem, since few databases are bound to their own dedicated named cache, unless the transaction logs are bound to a dedicated log cache (later topic), the transaction logs are using the default data cache. A common configuration for the default data cache is either all of it is at the server page size or some of it has been allocated to an 8x factor buffer pool (16K for a 2K server). Consequently, log IO's are forced to use the 2K IO size for flushes from the ULC to the primary log cache and consequently for log writes. Even when a 4K pool is available, the 4K pool is not exclusively reserved for the transaction log and other data pages push the log out of cache resulting in physical reads to perform checkpoints, log scans for triggers or other operations.

When tuning the ULC and log IO size, the three main considerations are:

- Log IO size (sp_logiosize)
- ULC Flush reasons
- Session tempdb user log cache size

The latter feature is especially helpful if tempdb log contention is contributing to system performance degradation – or if it appears like the tempdb log device is slow. These are describe in more detail in the following paragraphs.

sp_logiosize

The first step is to set the log IO size correctly. This requires a bit of analysis as you will need to find out the average transaction size and amount of concurrency within the system. As was described earlier, if the system does all atomic inserts and has log concurrency, any increase in the log IO size above the server page size is simply ineffective. The default of 2 log pages may

also be woefully inadequate if there are a lot of inserts into wide tables, especially if using micro-batching, decently sized transactions or in cases of high concurrency.

One of the first things to look at is the number of log writes vs. writes to the log (space allocations). Consider the following two snippets from two different servers one a 52 engine ASE and the other a 32 engine ASE (different transaction profiles):

```
-- 52 engine ASE
…
  Total ULC Flushes              4757.0          1.7      8624392
  ULC Log Records               22288.1          7.8     40408318       n/a

  Transaction Log Writes         1020.0          0.4      1849241       n/a
  Transaction Log Alloc          1518.9          0.5      2753780       n/a
  Avg # Writes per Log Page         n/a          n/a      0.67482       n/a


-- 32 engine ASE
…
  Total ULC Flushes              8438.7          3.7       506319
  ULC Log Records               19941.6          8.8      1196495       n/a

  Transaction Log Writes         1699.2          0.8       101949       n/a
  Transaction Log Alloc           726.2          0.3        43573       n/a
  Avg # Writes per Log Page         n/a          n/a      2.33973       n/a
```

'Transaction Log Writes' actually is the number of physical writes to the transaction log – i.e. log pages flushed to disk. 'Transaction Log Alloc' refers to the number of new transaction log pages allocated. By comparing the two, we can get a rough idea of how often full log pages are written to disk – and in a sense then if the log IO size is set too high. If you think about it, if a full log page is flushed to disk, then a new log page will need to be allocated – so it would be a 1:1 ratio. However, if a log page is written to disk that is not full, then a new log page does not need to be allocated. As a result, the number of log writes will be higher than the number of log pages allocated. Conversely, if the log IO size is 4K and the log writes happen at the 4K boundary (vs. being flushed early), then a single log write will occur – but two log pages will be allocated – a ratio of 1:2. This comparison is done via 'Avg # of Writes per Log Page'. According to the documentation:

> *"Avg # Writes per Log Page" reports the average number of times each log page was written to disk. The value is reported in the "count" column.*
>
> *In high throughput applications, this number should be as low as possible. If the transaction log uses 2K I/O, the lowest possible value is 1; with 4K log I/O, the lowest possible value is .5, since one log I/O can write 2 log pages.*

In other words, the 52 engine server is fairly close to the ideal 0.5 log writes considering a 4K log IO size. The 32 engine server is suffering from more than 2 physical writes for the same log page on average. Another value to consider is the number of log records per log page. In the above, the answer is ~14.7 for the 52 engine server and ~27.5 for the 32 engine server. While this is an average that can be tremendously skewed in 12.5 due to empty transactions, the one consideration of this is that unless the average transaction exceeds 10-14 rows in the log, a 2K ULC is likely not that effective.

There is a bit of a caution to be considered in setting the log IO size. Most systems generally are set so that the ULC size and sp_logiosize are set identically. While this generally is a good rule of thumb, if group commit sleeps are being effective at all, the size of the ULC in comparison to the log IO size is not a hard rule and should be consider in relation to the reasons the ULC is being flushed. For example, in the sp_sysmon at the beginning of the next section, the log IO size seems to be appropriately set at 4K – but the ULC size of 4K seems too large as it

rarely (<10%) gets filled. Of course, if the normal usage is 3K, then the results would be the same, so you do have to consider the problem of only measuring when filled vs. normal utilization. The best rule of thumb is likely that the ULC and the log IO size should be close to each other. A 16K ULC is rendered ineffective with a 2K log IO size assuming that the primary ULC flush reasons are either due to full ULC or commit and the typical ULC usage is considerably higher than 2K. Similarly, a 16K log IO size is useless when the largest transaction is only 1 row – unless concurrency is high enough that group commit sleeps are fairly high.

ULC Flush Reasons

Before changing the database log IO size, however, you should check the ULC flush reasons. Most DBA's are familiar with this section of the sp_sysmon report (repeated from earlier):

```
-- sp_sysmon fragment from a 12.5.3 server

Transaction Management
----------------------

  ULC Flushes to Xact Log         per sec      per xact       count  % of total
  -------------------------  ------------  ------------  ----------  ----------
    by Full ULC                     356.4           0.1      643933       8.5 %
    by End Transaction             1198.7           0.5     2165968      28.6 %
    by Change of Database            27.2           0.0       49073       0.6 %
    by Single Log Record           1047.3           0.4     1892392      25.0 %
    by Unpin                         34.2           0.0       61745       0.8 %
    by Other                       1520.3           0.6     2747156      36.3 %
  -------------------------  ------------  ------------  ----------
  Total ULC Flushes                4183.9           1.6     7560267

  ULC Log Records                 15172.8           5.8    27417193       n/a
  Max ULC Size During Sample          n/a           n/a           0       n/a

  ULC Semaphore Requests
    Granted                       37304.1          14.2    67408447     100.0 %
    Waited                            1.3           0.0        2284       0.0 %
  -------------------------  ------------  ------------  ----------
  Total ULC Semaphore Req         37305.3          14.2    67410731

  Log Semaphore Requests
    Granted                        3134.3           1.2     5663684      99.8 %
    Waited                            4.9           0.0        8816       0.2 %
  -------------------------  ------------  ------------  ----------
  Total Log Semaphore Req          3139.2           1.2     5672500

  Transaction Log Writes            695.9           0.3     1257528       n/a
  Transaction Log Alloc            1031.3           0.4     1863488       n/a
  Avg # Writes per Log Page           n/a           n/a     0.67482       n/a

  Tuning Recommendations for Transaction Management
  -------------------------------------------------
  - Consider decreasing the 'user log cache size'
    configuration parameter if it is greater than the
    logical database page size.
```

Before we analyze the above, note that this was from a 12.5.3 server vs. an ASE 15.x server. One way to tell is that the number of ULC flushes due to 'Single Log Record' are high. Overall, consider the following explanations for the log flush reasons:

by Full ULC – This is obvious – the number times a user's ULC was flushed because the transaction was larger than the ULC size configured for the server.

by End Transaction – While this appears obvious, i.e. the number of times the ULC needed to be flushed because of a commit (or rollback), in 12.5.x especially, this number could be artificially inflated as ASE 12.5.x and prior flushed empty user transactions caused by isolation level 3 reads or other empty transactions to the transaction log.

by Change of Database – Less obvious, when flushing the ULC to the actual primary log cache, the entire ULC needs to be flushed. If a user's transaction spans databases, the ULC has to be flushed before log records are written into the ULC for the new database context. Remember, IO's in ASE are done on a per-page basis, consequently, because each transaction log in each database are separate entities, the log pages in the user's ULC has to be flushed individually per database. A common cause of this in 12.5.x and prior releases was writes to tempdb – now replaced by a separate session tempdb ULC.

by Single Log Record – (aka System Log Record) ASE creates an SLR for any allocation page (OAM) modification in databases that have mixed data & log allocations (i.e. tempdb). SLR's force a flush of the ULC immediately to avoid potential internal deadlocks during "unpinning". Unlike ASE 12.5x, ASE 15.x no longer does an ULC flush in databases that do not need recovery (tempdb, reduced durability databases (15.5), etc.)

by Unpin – To speed up the processing of rollbacks, a modified datapage is "pinned" to the user's ULC that modifies it. If the user's transaction rollsback while still in the ULC, a larger log scan to identify all the records to be undone is avoided. If another user modifies the same page (regardless of row), the page has to be unpinned from the first user which forces a log flush of their ULC (assumes their transaction has not committed or else that would have flushed their ULC and unpinned the page anyhow). After a ULC flush has occurred, any rollback would require a log scan. ASE 15.x does not pin pages in databases that do not need recovery (tempdb, reduced durability databases).

Unfortunately, this is measured server wide and as a result, the averages reported above can be skewed by the checkpoint process (which does use ULC) as well as lot of small transactions (such as isolation level 3 selects or empty transactions in 12.5) mix with large transactions. Consequently, the above should be compared to the same information presented in monProcessActivity:

```
create existing table monProcessActivity (
        SPID                            int,
        InstanceID                      tinyint,
        KPID                            int,
        ServerUserID                    int,
        CPUTime                         int,
        WaitTime                        int,
        PhysicalReads                   int,
        LogicalReads                    int,
        PagesRead                       int,
        PhysicalWrites                  int,
        PagesWritten                    int,
        MemUsageKB                      int,
        LocksHeld                       int,
        TableAccesses                   int,
        IndexAccesses                   int,
        TempDbObjects                   int,
        WorkTables                      int,
        ULCBytesWritten                 int,
        ULCFlushes                      int,
        ULCFlushFull                    int,
        ULCMaxUsage                     int,
        ULCCurrentUsage                 int,
        Transactions                    int,
        Commits                         int,
        Rollbacks                       int,
)
```

```
external procedure
at "@SERVER@...$monProcessActivity"
go

…
```

There are two aspects to consider here. First, is the log I/O size itself; and second is any extra contention on the log semaphore as a result of the smaller ULC size or more frequent flushes. The latter is much more important than the former. The key to the log semaphore contention is to remember that it is more like a lock vs. a spinlock – processes that need to wait for the semaphore are put to sleep (monProcessWaits.WaitEventID=54). As a result, rather than using the server-wide averages as reported in sp_sysmon, you should instead look at the ULC flush reasons and AppendLogWaits by database in the MDA tables above as well as monProcessWaits. If time critical processes are waiting on the log semaphore frequently (monProcessActivity.Transactions vs. monProcessWaits.WaitEventID=54) and a good portion of their log flushes are due to ULC full (e.g. >30%), you may wish to increase the ULC size server wide to try to reduce the log semaphore contention. On the other hand, if most of the contention is on tempdb, you may need to increase the session ULC size (below) or consider multiple tempdb's (later discussion).

In the above example, once upgraded to ASE 15, most of the ULC flushes should disappear that are attributable to Single Log Record – leaving just 'by Other' and 'by End Transaction' at about 1,100 per second each and 'by Full ULC' at ~350 per second. Even though this would only be about 15%, it is in the grey area where close look at the MDA tables would be warranted to see if this 15% is the more critical processes.

Additionally, as a separate case, if the primary use of the system is for a reduced durability database with minimal logging for DML, the ULC size may need to be increased as if it were a session tempdb ULC (see next section).

Session Tempdb Log Cache

In the above section, one of the causes of a premature ULC flush was identified as being due to 'by Change of Database'. While ASE 15 does add a separate 'session tempdb log cache size' to create a separate ULC for user tempdb activities – and while it does help alleviate this problem – it is not the primary reason for a private tempdb ULC for each user.

In a large number of customers, in a questionably effective attempt to reduce tempdb system table contention, temporary tables were often created at the beginning of the procedure and then populated using normal logged insert/select, update or delete operations. These logged operations could often be characterized by the following:

- Many of the DML operations were bulk set operations that affected all or multiple rows in the temp table in a single statement.
- Multiple DML operations often were executed as atomic transactions as the initial insert/selects and subsequent updates are deletes were outside the scope of transactions.

The latter was more an issue with ASE 12.5 as each of these would be written to the tempdb transaction log, causing log contention within tempdb.

Remember, the ASE transaction log is used for two primary purposes: 1) it guarantees recovery through write-ahead logging; and 2) it is used to provide rollback support for transactions that don't fit in the ULC or were prematurely flushed due to "pinning" or other reason. The first is unnecessary for tempdb as recovery isn't a consideration, and the second wouldn't occur much at all in tempdb as #temp tables are not shared between sessions.

In ASE 15.0 (for tempdb only) and 15.5 (RDDB), the ULC will not be written to the transaction log if the transaction fully fits in the ULC and is either committed or rolled-back while fully cached in the ULC. There are other constraints for RDDB that require logging in any case (e.g. database is replicated), but primarily this is the implementation for minimally logged DML. This has some interesting aspects to consider from an application perspective:

- Minimally logged operations such as select/into may be able to manipulate considerable rows and avoid logging completely if the allocation pages and other aspects that are logged fit completely in the tempdb ULC.
- Atomic inserts, updates or deletes into tempdb would avoid logging entirely (for temporarily holding application data and then modifying the production database with an insert/select or update with join).

To exploit this behavior, a separate ULC log cache just for the user's assigned tempdb was added in ASE 15 – called the 'session tempdb log cache'. The separate log cache also was created to reduce the number of times the primary log cache was flushed due to a change of the database context, otherwise simply increasing the existing ULC would have the same effect.

Remember, however, that the best efficiency of PLC flushing is if the log cache uses the same size buffer pool as the log I/O size as set via sp_logiosize. The best option, then, is to ensure that the log IO size and the ULC are set appropriately. For reduced durability databases such as tempdb, using a 8x factor (e.g. 16K) buffer pool for the log may now benefit as smaller transactions won't be hitting the log and as a result, the primary logging activity will be larger transactions. Note that for tempdb, this may not be persistent as the database is recreated from model each time (as would be in-memory databases). Consequently, along with dbcc tune(des_bind) in the server reboot script, you may need to add sp_logiosize for tempdb.

Transaction Micro-Batching/Exploiting the ULC

Above in discussing the log group commit sleeps, one of the reasons given for a log flush was if the process that was waiting for a log write to happen arrived at the head of the run queue (to be woken to see if the IO had completed). For example, consider an single process that is doing a lot of atomic single row inserts – while all other users on the system are simply issuing queries. Since the 'writer' process is the only one writing to the transaction log, when ASE sees it at the head of the run queue, rather than continue waiting for a different process to complete the log page, ASE simply flushes it to disk and allows the process to commit. The net result is that every atomic insert has to wait for the log page to be flushed to disk.

One way to avoid this is to simply exploit the ULC to hold enough log records to fill one or more log pages completely and flush them to the transaction log in small batches. Consider the following chart:

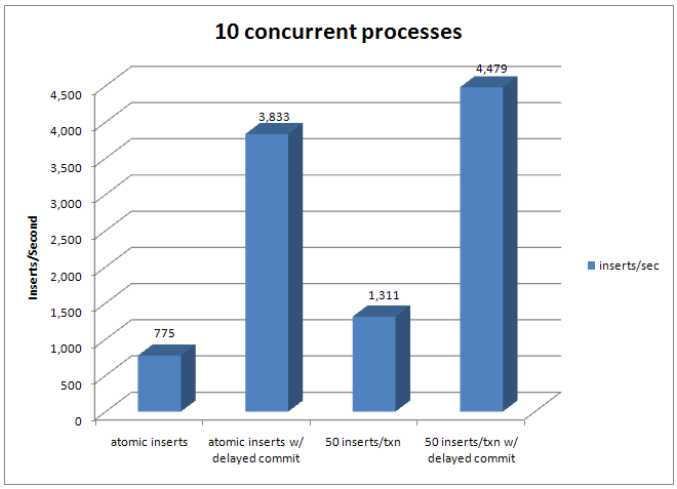


Figure 11 – Comparison of Atomic Inserts and Explicit Transaction Grouping

What delayed commit does will be described in more detail later. However, note that there is about a 400% improvement when using it – even when grouping 50 inserts into a single transaction. What delayed commit does essentially is avoid the group commit sleep and the log write to disk. Based on the above, given a typical ULC (8K for a 4K server as this was) and a typical medium width table (~20 columns), each transaction group was likely filling multiple log pages per commit – consequently, it can be inferred that the group commit sleep was the minor part of the log waits and that the device speed the predominant portion as described earlier. Regardless, notice that without delayed commit, the increase in throughput was about 70% and that with delayed commit, the increase was about 25% - both appreciable numbers – but still far below what is often desirable.

Now, consider the following graph of 10 threads performing high speed inserts in ASE 15.0.3 in a similar test:

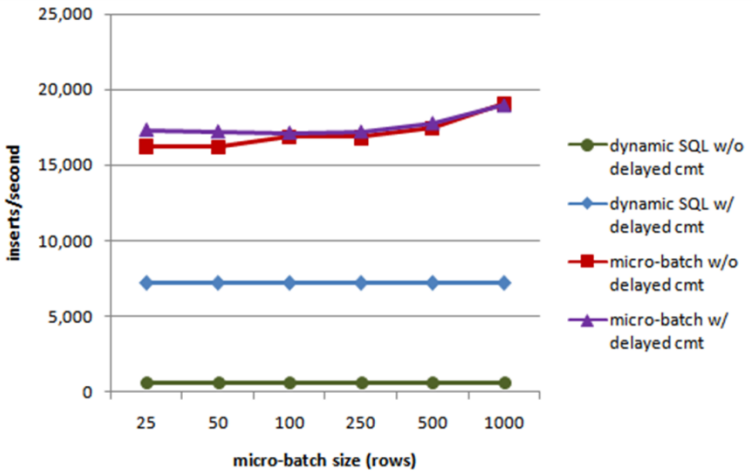


Figure 12 – Comparison of Atomic Inserts and Micro-Batching

The average throughput was ~1000 inserts per second – close to the above. By adding in delayed commit to avoid the group commit sleeps and disk write time, it jumped to ~7,500 inserts per second. By using micro-batching, it ranged from ~17,000 inserts per second to nearly 20,000 inserts per second at a 1,000 row batch size – and delayed commit had little to no effect.

Micro-batching is implemented by using the bulk API features of Sybase OpenClient or jConnect (for Java/JDBC) using the fully logged/transaction form of the bulk interface (vs. the minimally logged form). While dynamic SQL by itself eliminates much of the optimization cost of executing inserts by using fully prepared statements, since it uses the RPC protocol stream, it cannot batch together multiple inserts to leverage network efficiency. Consequently each insert statement is a complete network operation. By using the bulk interface, multiple rows can be transferred at once using array inserts, and are logged (in this case) much more easily by transferring from the servers bulk insert arrays directly into the transaction log. To implement this in java using JDBC, you need to make sure that the connection property 'ENABLE_BULK_LOAD'=true and the connection property 'DYNAMIC_PREPARE'=true (enables fully prepared statements vs. just client-side/API prepared statements) and then use the standard JDBC preparedStatement() and addBatch() methods.

To do this, however, may require an application rewrite for stream handlers or other processes that perform a lot of insert activity to batch up the inserts until either a small batch has been achieved or a timer has expired. While larger batch sizes do improve the throughput (as indicated by the tailing up at the end), the problem is that it also increases the response time for the first row that was queued to be inserted. Consequently, these small "micro-batches" optimize the efficiency while keeping the overall response time to any single event small. Consider the following table listing the per second throughput vs. response time:

Scenario	Total Throughput	Threads	Rate/ Thread	Batches/ Thread	Single Row Response (ms)
Atomic inserts	1000	10	100	100	10
50 row batch	17000	10	1700	34	29
100 row batch	17000	10	1700	17	59
250 row batch	17000	10	1700	7	143
500 row batch	18000	10	1800	4	250
1000 row batch	19000	10	1900	2	500

The maximum size of the batch to use would be determined by the maximum acceptable response time from the application.

Asynchronous Log Service

In order to ensure the full ACID durability, most DBMS's use 'blocking commits' to ensure that a committed transaction is fully on disk and recoverable before the commit() method or commit SQL statement returns control to the application. The term 'blocking commit' is used to note that it is a synchronous/blocking operation vs. an asynchronous operation. The normal sequence for log writing was depicted earlier in this section. This changes when using the ALS to the following:

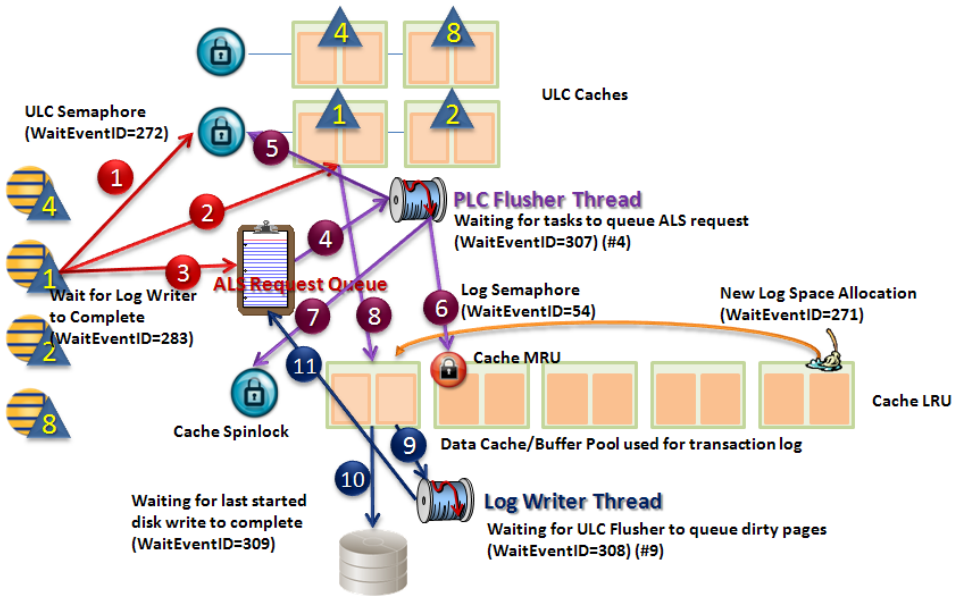


Figure 13 – Transaction Logging with ALS Enabled

While this looks complicated, it actually is not that different from earlier, except we now have a few additional threads and they need to coordinate via the spinlocks.

1. When the SPID needs to write modified row images to its ULC, it first needs to grab the ULC spinlock as before.
2. The SPID then writes to the ULC as before
3. When the SPID's ULC is full or the transaction commits, the SPID waits until it can get access to the ALS request queue for the PLC flusher and appends its request (FIFO) for a ULC flush. It then goes to sleep to wait for the Log Writer to complete (WaitEventID=283) which it will note when its request is completed.
4. The PLC flusher thread polls each database with ALS enabled looking for log append requests. This idle polling loop is WaitEventID=307 'Waiting for tasks to queue ALS requests'
5. When it gets a request, the first thing the PLC Flusher thread needs to do is grab the ULC spinlock to prevent the SPID from making further changes to the ULC.
6. Then the PLC flusher grabs the log semaphore – since it is the only process trying to get the semaphore (other than the checkpoint), contention is non-existent
7. Then the PLC Flusher grabs the cache spinlock – much like the user SPID used to in normal logging.
8. Then the PLC Flusher flushes the ULC for the SPID whose request it is currently processing.
9. The Log Writer thread is normally in a wait loop (WaitEventID=308) "Waiting for the ULC Flusher to queue dirty pages"
10. The Log Writer thread writes the dirty log pages to disk – note that the WaitEventID is not 55 – but rather 309 – "Wait for the last started disk write to complete"
11. The Log Writer then marks the log append request complete

As suggested in the above, when ALS is enabled, there are two new system threads running in ASE – the Log Writer and the PLC Flusher. These two threads handle all the log writing requirements for *all* the databases with 'async log service' enabled via sp_dboption. This is a bit of a limitation – if multiple databases have ALS enabled, the two new threads have to process all of the databases. On the plus side, the threads will run at a much higher priority – a priority of '3'. Note that this is a system (kernel) level priority as the highest user priority is '4' (EC1).

As with any performance feature, used correctly, ALS can provide a considerable boost in throughput. Equally, however, used incorrectly, it can severely degrade performance.

ALS – Very Large SMP/High Volume OLTP

The asynchronous log service was implemented in ASE 12.5.0 for very large SMP systems with high transaction volumes in which the log semaphore contention was much higher and longer than the log write delays simply due to the concurrency. It was first used in TPC-C benchmarks on HP Superdome machines with 64 engines running ASE and provided ASE 12.5 the boost to go from ~400K TPM to over 500K TPM. This type of environment is the epitome when ALS is most effective:

- Large to very large SMP (20+ engines)
- Short, high volume OLTP transactions
- High log semaphore contention
- High speed transaction log devices

Consider the following sp_sysmon output from a 32-engine server:

```
  Task Context Switches Due To:
    Voluntary Yields              1117.3          0.5       67038       3.4 %
    Cache Search Misses             23.6          0.0        1418       0.1 %
    Exceeding I/O batch size         4.1          0.0         243       0.0 %
    System Disk Writes             490.9          0.2       29452       1.5 %
    Logical Lock Contention         87.9          0.0        5272       0.3 %
    Address Lock Contention          6.8          0.0         408       0.0 %
    Latch Contention              1278.9          0.6       76734       3.9 %
    Log Semaphore Contention      3645.1          1.6      218705      11.0 %
    PLC Lock Contention            437.1          0.2       26228       1.3 %
    Group Commit Sleeps           4158.0          1.8      249477      12.5 %
    Last Log Page Writes           975.2          0.4       58514       2.9 %
    Modify Conflicts              1007.1          0.4       60428       3.0 %
    I/O Device Contention            0.0          0.0           0       0.0 %
    Network Packet Received       5039.8          2.2      302390      15.2 %
    Network Packet Sent           5736.3          2.5      344179      17.3 %
    Network services              9358.4          4.1      561505      28.2 %
    Other Causes                  -234.7         -0.1      -14083      -0.7 %

…

Transaction Profile
-------------------

  Transaction Summary            per sec      per xact       count  % of total
  -------------------------  ------------  ------------  ----------  ----------
    Committed Xacts               2262.8           n/a      135765     n/a
…
  =========================  ============  ============  ==========
    Total Rows Affected           6368.9           2.8      382136

…

Transaction Management
----------------------

  ULC Flushes to Xact Log         per sec      per xact       count  % of total
  -------------------------  ------------  ------------  ----------  ----------
```

```
  by Full ULC                       9.1            0.0          548       0.1 %
  by End Transaction             3408.2            1.5       204492      40.4 %
  by Change of Database          3390.9            1.5       203453      40.2 %
  by Single Log Record             79.3            0.0         4758       0.9 %
  by Unpin                       1547.5            0.7        92850      18.3 %
  by Log markers                    3.6            0.0          218       0.0 %
-------------------------  ------------  ------------  ----------
Total ULC Flushes                8438.7            3.7       506319

ULC Flushes Skipped              per sec       per xact        count  % of total
-------------------------  ------------  ------------  ----------  ----------
  by PLC Discards                  0.2            0.0           14     100.0 %
-------------------------  ------------  ------------  ----------
Total ULC Discards                 0.2            0.0           14

ULC Log Records                19941.6            8.8      1196495       n/a
Max ULC Size During Sample           n/a            n/a            0       n/a

ULC Semaphore Requests
  Granted                      42265.7           18.7      2535939      99.0 %
  Waited                         437.1            0.2        26228       1.0 %
-------------------------  ------------  ------------  ----------
Total ULC Semaphore Req        42702.8           18.9      2562167

Log Semaphore Requests
  Granted                       3902.7            1.7       234161      51.7 %
  Waited                        3645.1            1.6       218705      48.3 %
-------------------------  ------------  ------------  ----------
Total Log Semaphore Req         7547.8            3.3       452866

Transaction Log Writes          1699.2            0.8       101949       n/a
Transaction Log Alloc            726.2            0.3        43573       n/a
Avg # Writes per Log Page            n/a            n/a      2.33973       n/a
```

Compared to the 50 engine example from earlier, we can see a huge difference in log semaphore contention. For example, in the task context switching section, consider the following:

```
-- 52 engine ASE at 70-80% cpu usage
  Task Context Switches Due To:
…
    Log Semaphore Contention        11.6            0.0        20977       0.0 %
    PLC Lock Contention              1.9            0.0         3503       0.0 %
    Group Commit Sleeps            126.1            0.0       228583       0.5 %
    Last Log Page Writes           257.5            0.1       466862       1.0 %
…

-- 32 engine ASE at 60% cpu utilization
Task Context Switches Due To:
…
    Log Semaphore Contention       3645.1            1.6       218705      11.0 %
    PLC Lock Contention             437.1            0.2        26228       1.3 %
    Group Commit Sleeps            4158.0            1.8       249477      12.5 %
    Last Log Page Writes            975.2            0.4        58514       2.9 %
…
```

Note the difference?? In the larger SMP machine, the log semaphore contention is almost negligible – at 11.6 occurrences per second. On the smaller system, it accounts for one of the larger reasons for tasks context switching – and is happening at an average of nearly 4x per millisecond. Note that this section of the sp_sysmon report merely lists what is causing tasks to get bumped off the CPU. The actual amount of contention is measured later in the report with the following lines:

```
-- 52 engine ASE at 70-80% utilization
…
  Log Semaphore Requests
    Granted                       4203.1            1.5      7620299      99.7 %
    Waited                          11.6            0.0        20977       0.3 %
  -------------------------  ------------  ------------  ----------
  Total Log Semaphore Req         4214.7            1.5      7641276
…

-- 32 engine ASE at 60% cpu utilization
…
  Log Semaphore Requests
    Granted                       3902.7            1.7       234161      51.7 %
    Waited                        3645.1            1.6       218705      48.3 %
  -------------------------  ------------  ------------  ----------
```

```
  Total Log Semaphore Req          7547.8           3.3      452866
```

On a per second basis, the 32 engine ASE has nearly double the log semaphore requests as does the 52 engine ASE – and of those requests, nearly half of them have to wait. Let's take a look now at the log writes from each:

```
-- 52 engine ASE
  Task Context Switches Due To:
…
    Group Commit Sleeps             126.1            0.0      228583       0.5 %
    Last Log Page Writes            257.5            0.1      466862       1.0 %
…
  Transaction Summary            per sec      per xact       count  % of total
  -------------------------  ------------  ------------  ----------  ----------
    Committed Xacts              2854.9            n/a     5175863     n/a
    Total Rows Affected         24695.9            8.7    44773665
  Total ULC Flushes              4757.0            1.7     8624392
  ULC Log Records               22288.1            7.8    40408318       n/a


  Transaction Log Writes         1020.0            0.4     1849241       n/a
  Transaction Log Alloc          1518.9            0.5     2753780       n/a


-- 32 engine ASE
  Task Context Switches Due To:
…
    Group Commit Sleeps            4158.0            1.8      249477      12.5 %
    Last Log Page Writes            975.2            0.4       58514       2.9 %
…
  Transaction Summary            per sec      per xact       count  % of total
  -------------------------  ------------  ------------  ----------  ----------
    Committed Xacts              2262.8            n/a      135765     n/a
    Total Rows Affected          6368.9            2.8      382136
  Total ULC Flushes              8438.7            3.7      506319
  ULC Log Records               19941.6            8.8     1196495       n/a


  Transaction Log Writes         1699.2            0.8      101949       n/a
  Transaction Log Alloc           726.2            0.3       43573       n/a
```

Notice that the 32 engine ASE has about double the total number of ULC flushes but actually fewer log records being flushed per second – despite the slightly higher number of log records per transaction. As a result, it is flushing fewer log records from the ULC to the primary log per ULC flush than the 52 engine server – as we saw earlier, this results in more physical log writes. However, we need to make sure that the log device is not causing the delays. Unfortunately, this customer did not provide MDA data which would have been easier to determine the degree of contention as well as the log write throughput. Based on the sp_sysmon's last log page writes, it would appear not – and double checking the log devices in sp_sysmon, we can see the following:

```
  Device:
    /dev/rvgds_mbc_log
    mbc_log                      per sec      per xact       count  % of total
  -------------------------  ------------  ------------  ----------  ----------
    Reads
      APF                           0.0            0.0           0       0.0 %
      Non-APF                       1.0            0.0          60       0.1 %
    Writes                        907.3            0.4       54439      99.9 %
  -------------------------  ------------  ------------  ----------  ----------
  Total I/Os                      908.3            0.4       54499      38.9 %

…

  Device:
    /dev/rvgds_tos_log
    tos_log                      per sec      per xact       count  % of total
  -------------------------  ------------  ------------  ----------  ----------
    Reads
      APF                           0.0            0.0           0       0.0 %
      Non-APF                       1.8            0.0         110       0.2 %
    Writes                        796.4            0.4       47783      99.8 %
  -------------------------  ------------  ------------  ----------  ----------
  Total I/Os                      798.2            0.4       47893      34.2 %
```

So a total of about 1700 writes per second – completely in line with the sp_sysmon line with respect to the number of log writes. In either case, each log device is achieving slightly more than 1ms per IO – so, while the devices could be faster, they are fast enough that unless you implement an SSD, it is not likely to improve the speed. Because of their speed, this implies that the predominant cause of log semaphore contention is just the high concurrency and frequent access to the log semaphore. As a result, this is a case in which ALS is a likely candidate.

When not to use ALS

While the documentation states that ALS cannot be used with fewer than 4 engines, the primary purpose of this feature was to eliminate spinlock contention on the log semaphore when using very large numbers of engines (such as 20-30) and especially in very large SMP environments of 40 or more engines – and when attempting very high OLTP transaction rates. Further, since the log semaphore is held by the process writing to the log until the physical log writes complete, the physical log writes to disk must be first eliminated as the major contributor to the log semaphore contention. This can be accomplished by verifying that WaitEventID=55 (log physical writes) is not a significant percentage of the Waits of WaitEventID=54 (log semaphore contention). Finally, the correct database has to be identified to determine which one to enable the ALS option on.

One of the limitations of sp_sysmon is that it does not identify anything beyond the log semaphore contention – not which databases, not the underlying cause (disk writes), etc. In addition, it only looks at log semaphore contention and does not measure when ALS is a bigger problem. Consequently, after enabling ALS it can appeared as if it resolved the issue, when in fact all that it appears to have done is traded the log semaphore contention for a bigger issue with ALS caused waits. Consider the following system wait events from a benchmark system with only 10 engines but with ALS enabled.

Wait Event ID	WaitEvent	Waits	WaitTime (sec)	Comments
214	waiting on run queue after yield	44,577,448	7	High cpu contention due to high logical reads
215	waiting on run queue after sleep	37,305,361	12	CPU waits due to log writer, locking, network
55	wait for i/o to finish after writing last log page	8,016,604	4.42	Waiting for log device IO to return
250	waiting for incoming network data	5,238,702	3406.21	Awaiting for command from client
41	wait to acquire latch	3,632,235	34.39	Latch contention on tb_ctc/tb_ctc_frs??
150	waiting for a lock	2,090,361	71.65	Many due to replay missed commit
251	waiting for network send to complete	1,720,097	8.84	Packet size is causing this
51	waiting for last i/o on MASS to complete	1,377,110	0.9	Standard disk write
308	Waiting for ULC Flusher to queue dirty pages.	1,351,424	123.89	ALS
36	waiting for MASS to finish writing before changing	1,262,744	1.09	Standard disk write
52	waiting for i/o on MASS initated by another task	920,520	0.94	Disk write by HK or chkpt blocking users
29	waiting for regular buffer read to complete	715,731	6.53	Physical Reads
31	waiting for buf write to complete before writing	714,392	0.93	HK or chkpt waiting on user mod in data cache
54	waiting for write of the last log page to complete	617,854	1.28	ALS
307	Waiting for tasks to queue ALS request.	605,121	124.83	ALS
283	Waiting for Log writer to complete	603,578	1.98	ALS
272	waiting for lock on ULC	388,614	1.02	PLC contention (related to ALS)
124	wait for mass read to finish when getting page	114,694	0.52	Physical APF (large IO)
309	Waiting for last started disk write to complete	60,943	0.04	ALS initiated disk write

Wait Event ID	WaitEvent	Waits	WaitTime (sec)	Comments
171	waiting for CTLIB event to complete	33,711	0.15	CIS proxy queries or RepAgent
53	waiting for MASS to finish changing to start i/o	16,736	0	
178	waiting while allocating new client socket	10,525	125.77	New client login/connection
35	waiting for buffer validation to complete	289	0	
37	wait for MASS to finish changing before changing	11	0	
83	wait for DES state is changing	5	0	

In the above, the system is waiting more on the log device than anything else transaction log related – in which case, ALS is of no use at all. Now, consider the following which is system wait events from a similar 10 engine production system also with Asynchronous Log Service enabled:

Wait Event ID	WaitClass	WaitEvent	Waits	Wait Time (sec)
214	waiting to be scheduled	waiting on run queue after yield	121,789,298	13.76
215	waiting to be scheduled	waiting on run queue after sleep	74,582,040	42.84
250	waiting for input from the network	waiting for incoming network data	25,255,516	7323.82
29	waiting for a disk read to complete	waiting for regular buffer read to complete	4,147,034	44.58
283	waiting on another thread	Waiting for Log writer to complete	3,757,906	81.50
308	waiting on another thread	Waiting for ULC Flusher to queue dirty pages.	2,990,285	21.45
307	waiting on another thread	Waiting for tasks to queue ALS request.	2,203,541	18.78
251	waiting to output to the network	waiting for network send to complete	1,592,016	3.02
150	waiting to take a lock	waiting for a lock	904,866	40.48
36	waiting for memory or a buffer	waiting for MASS to finish writing before changing	840,101	4.25
31	waiting for a disk write to complete	waiting for buf write to complete before writing	648,305	7.93
171	waiting to output to the network	waiting for CTLIB event to complete	441,793	2.93
309	waiting for a disk write to complete	Waiting for last started disk write to complete	403,446	0.90
55	waiting for a disk write to complete	wait for i/o to finish after writing last log page	399,197	1.15
51	waiting for a disk write to complete	waiting for last i/o on MASS to complete	367,477	2.04
52	waiting for a disk write to complete	waiting for i/o on MASS initated by another task	179,598	1.20
124	waiting for internal system event	wait for mass read to finish when getting page	86,051	0.83
272	waiting for internal system event	waiting for lock on ULC	59,703	3.15
41	waiting for internal system event	wait to acquire latch	45,981	3.94
169	waiting for internal system event	wait for message	36,369	27.15

In the case of the production system, one of the top 5 causes of wait events (ignoring 250) is the ALS. To really see how this is impacting the system, we need to get an idea of which SPIDs are the ALS processes. Consider the following table of system SPIDs in the production system:

SPID	Command	CpuTime	Wait Time	Physical Reads	Logical Reads	Physical Writes
2	DEADLOCK TUNE	0	143,719,300	0	0	0
3	CHECKPOINT SLEEP	99,000	143,055,600	730	1,281,223	2,806,516
4	HK WASH	3,493,400	139,843,200	205	7,335	21,304,680
5	HK GC	6,200	143,187,600	628	34,057	5,817
6	HK CHORES	549,300	142,612,200	706	993,493	232,874
7	AUDIT PROCESS	26,200	143,166,100	19	69,572	157,637
8	PORT MANAGER	0	143,719,300	0	0	0
9	NETWORK HANDLER	6,200	143,180,800	0	0	0
10	NETWORK HANDLER	30,300	143,161,600	0	0	0
11	NETWORK HANDLER	300	138,306,400	0	0	0
14	PLC FLUSHER	8,298,100	134,936,900	1,126	40,188	0
15	LOG WRITER	11,683,500	131,715,000	0	0	6,464,505
39	REP AGENT	5,800	143,188,800	28	38,885	2,939

SPID	Command	CpuTime	Wait Time	Physical Reads	Logical Reads	Physical Writes
40	LICENSE HEARTBEAT	0	143,194,600	100	37,129	1
42	SITE HANDLER	100	143,154,100	0	6	0
608	SITE HANDLER	0	26,837,000	0	0	0

The highlighted rows are the system processes used by the ALS service. Consider the follow process level wait statistics (monProcessWaits) for the ALS SPIDs and some of the user processes. In each case, only the top wait event for each SPID is noted:

SPID	KPID	ServerUserID	WaitEventID	Waits	WaitTime	msPerWait
15	3145776	0	308	7,188,311	129,573,500	18
14	3080239	0	307	4,689,854	129,214,600	27
1015	492177098	14	283	55,507	1,211,100	21
123	492570321	14	283	53,312	1,202,800	22
155	491783875	14	283	40,108	1,188,500	29
796	492439247	14	283	42,055	1,175,700	27
519	493094619	14	283	40,609	1,154,600	28
1936	489490082	14	283	53,245	1,135,900	21
1460	493291229	14	283	37,978	1,132,600	29
410	492898007	14	283	36,199	1,129,400	31
1010	491849412	14	283	43,201	1,100,100	25
1439	492308173	14	283	45,962	1,093,600	23
280	491587263	14	283	43,450	1,088,500	25
2119	492046004	14	283	53,095	1,088,400	20
202	490997433	14	283	58,351	1,078,200	18
1672	491718337	14	283	39,159	1,076,900	27
485	493160156	14	283	34,764	1,071,000	30
1892	491390602	14	283	54,338	1,039,800	19
1970	491062970	14	283	41,509	1,035,800	24
1314	490210937	14	283	44,764	1,033,700	23
(…and so on…every non-system SPID with WaitEventID=283 as the highest)						

Essentially, the ALS Log Writer most of the time is waiting for the PLC flusher which is turn waiting on the users – who are waiting on access to the PLC flusher. The below table details the full wait events for the ALS processes as well as one of the user SPIDs in the above table (sorted by WaitTime):

SPID	KPID	Wait Event ID	Description	Waits	Wait Time	ms Per Wait	Comments
14	3080239	307	Waiting for tasks to queue ALS request.	4,689,854	129,214,600	27	Waiting on SPID to flush ULC to ALS request queue
14	3080239	150	waiting for a lock	191,690	4,442,800	23	Lock on PLC or ALS queue
14	3080239	36	waiting for MASS to finish writing before changing	320,664	676,700	2	Log cache flush blocked by log writer
14	3080239	214	waiting on run queue after yield	55,684,414	450,500	0	CPU contention (minor)
14	3080239	272	waiting for lock on ULC	2,724	90,800	33	PLC semaphore contention
14	3080239	29	waiting for regular buffer read to complete	1,126	16,700	14	Physical read of log pages due to ULC overrun?
14	3080239	31	waiting for buf write to complete before writing	120	100	0	
15	3145776	308	Waiting for ULC Flusher to queue dirty pages.	7,188,311	129,573,500	18	Waiting on PLC flusher
15	3145776	214	waiting on run queue after yield	187,887,626	1,151,500	0	CPU contention (minor)
15	3145776	309	Waiting for last started disk write to complete	415,997	929,600	2	Log flush to disk

SPID	KPID	Wait Event ID	Description	Waits	Wait Time	ms Per Wait	Comments
1015	492177098	250	waiting for incoming network data	64,923	22,120,100	340	Awaiting command
1015	492177098	283	Waiting for Log writer to complete	55,507	1,211,100	21	Waiting for ALS request queue access
1015	492177098	29	waiting for regular buffer read to complete	18,339	252,800	13	Physical read
1015	492177098	150	waiting for a lock	4,688	180,900	38	Logical lock on table/index
1015	492177098	41	wait to acquire latch	518	46,000	88	Latch on DOL index page
1015	492177098	36	waiting for MASS to finish writing before changing	5,423	17,900	3	Contention with HKGC or checkpoint or wash marker flush
1015	492177098	272	waiting for lock on ULC	455	17,700	38	PLC semaphore contention
1015	492177098	214	waiting on run queue after yield	2,452	7,300	2	CPU contention (minor)
1015	492177098	124	wait for mass read to finish when getting page	618	6,700	10	Physical APF (large IO) - likely table or index scan
1015	492177098	171	waiting for CTLIB event to complete	344	2,200	6	Waiting for CIS (or RepServer for RepAgent) response
1015	492177098	251	waiting for network send to complete	2,644	1,100	0	Send sleep
1015	492177098	55	wait for i/o to finish after writing last log page	1,113	1,000	0	Log write
1015	492177098	35	waiting for buffer validation to complete	5	400	80	
1015	492177098	51	waiting for last i/o on MASS to complete	330	300	0	Wash marker flush
1015	492177098	52	waiting for i/o on MASS initated by another task	63	200	3	HKGC or checkpoint contention
1015	492177098	31	waiting for buf write to complete before writing	30	100	3	HKGC or checkpoint contention
1015	492177098	54	waiting for write of the last log page to complete	2	0	0	Log semaphore contention

As highlighted in yellow, the PLC flusher is spending most of its time waiting for SPIDs to queue requests – about 27ms per wait on average. Once it receives requests, it then is waiting an average 23 milliseconds to get the lock on the PLC to be flushed or the ALS request queue. A 23 millisecond wait automatically limits throughput to 44 transactions/second at best – without considering the time to actually write to disk. The PLC flusher is hampered slightly by the ongoing logical reads driving CPU contention as the most frequent interruption to the PLC flusher is CPU access – although each wait is much less than 1 millisecond.

The Log Writer in turn is spending most of its time waiting on the PLC Flusher to queue up the dirty log pages by flushing the next PLC to the log cache. Again, it is hampered somewhat by the CPU contention – but mostly, the PLC Flusher and Log Writer threads are spending most of their time waiting for each other. The user tasks seem to be able to get their tasks on the request queue, but then spend most of their time waiting for the commit to be returned from the Log Writer – net effect is that everyone is waiting on everyone else, hampered somewhat when something happens by CPU contention due to high logical reads.

This is a clear case of when ALS is actually degrading performance and should be disabled. It is also interesting as sp_sysmon shows low semaphore contention at the same time the MDA tables show high log append waits – and that is the difference discussed earlier between the two. Log semaphore waits measures exactly that – while the log append waits measures any wait – whether log semaphore or whether ALS thread waits. While ALS might be useful in low-

mid range SMP environments such as this – it is extremely doubtful. Any decision to enable ALS with fewer than 20 engines should be viewed with skepticism, and if enabled, it should be carefully monitored to ensure that it does not make the situation worse. Consider the following rules of thumb about enabling ALS:

- At least 16-20 engines
- Log semaphore contention above 30% and high task context switching
- Log device performance at ~1ms/IO or better
- Log semaphore waits (WaitEventID=54) is ~10ms or more per wait

Consider disabling ALS when:

- Less than 20 engines
- Log writes per page is <1
- Normal processing waits on WaitEventID=283 is >10ms per wait
- PLC flusher waits on log events being queued (WaitEventID=307) or Log Writer waits on PLC flusher (WaitEventID=308) >10ms per wait

Delayed Commit

The concept of 'delayed commits' is a bit misnamed and should have been called non-blocking commits. By default, as suggested many times above, once a process issues the commit command, the commit method is a synchronous/blocking command from that session's viewpoint. It can't do anything else until the commit returns. For ASE, this means that the process has to wait until the log records (including the commit itself) are physically on disk.

The more concurrent atomic transactions that a system experiences, the more log semaphore contention will be experienced. The reason has less to do with the log semaphore and more to do with the speed of the log device itself. If you think about it, normally we consider that writes to a table are cached writes – in that we update the table in data cache. However, for atomic transactions affecting a single row, that row modification must be logged – and that logged row modification essentially means that for every row modified, there is a disk write required before the process can modify the next row. Unfortunately, unlike the data pages where concurrent inserts may be happening in different tables of different areas of the same table, with the log writes, everyone is serializing on that last log page. While group commit sleeps help overall throughput if users are modifying different rows or different tables, it doesn't help the response time for a single user as that user must wait for the physical disk write.

Both Sybase ASE and Oracle DBMS's support a flavor of non-blocking commits. In this case, the control is returned to the client process as soon as some particular stage of the commit processing is completed and it appears as if the rest will be successful. In the case of ASE, the delayed commit option provides this feature by returning control to the application as soon as the ULC is flushed to the primary log cache – the process does not wait for any subsequent group commit sleep or the time for the actual write to disk to occur.

This does not lead to the possibility of database corruptions or other non-recoverable issues. All that happens is that in the event of a system crash, *if* the IO for the last log page didn't

complete, then that page will be missing from the log and all transactions committed on that page will be rolled back. Obviously for larger transactions, this feature will have less of a performance boost as the ULC will buffer the pending log writes. However, for atomic transactions, the throughput increase can be 10-50x higher with this feature enabled. Considering that IO's to enterprise storage systems take ~2ms or less, the amount of data that could be lost given atomic transactions is extremely minimal.

The interesting aspect to delayed commit is that it can be enabled as a database option or at the session level. The latter is interesting as processes that could benefit from delayed commit can leverage it while those that cannot afford any data loss do not have to. Note that processes that do not have delayed commit enabled are extremely minimally impacted if at all, by executing after a process that does have it enabled. The sequence in that case would be similar to:

- SPID 1 writes to the transaction log with 'delayed commit' enabled. Rather than waiting for the IO to complete, it simply commits immediately, releasing the log semaphore.
- SPID 2 writes to the transaction log with 'delayed commit' disabled by appending its log records to the current log page.
- When SPID 2 is finished appending its log records, it flushes the pages to disk – and waits for the write success – just like it normally would.
- Since two IOs have been issued to the OS for the same log page, the SPID will have to wait until its IO completes. The time it takes depends on whether the OS, SAN or physical device consolidates the two writes into a single write through some write queuing mechanism (such as Linux's IO scheduler or SCSI/SATA 2 NCQ implementation).

The effect is that a subsequent SPID with delayed commit off will effectively force a flush for preceding log writes with delayed commit on – but at negligible or no additional overhead. The same SPID that performed the writes with delayed commit can force the flush simply by issuing a non-empty explicit transaction with delayed commit off (more on this below with respect to procs without recovery).

Delayed commit can be used safely to improve response time and scalability then any time:

- The database or session is processing non-business critical data (e.g. auditing or worktable data)
- Stored procedures or batch processes that have a lot of atomic SQL statements but are not executed within a transaction nor use explicit transactions.
- Applications that buffer previously executed transactions and can resubmit them if necessary.

Non-business critical data

While transaction logging is critical for the durability of business data, in a lot of cases, today's systems experience a lot of non-business data inserts or modifications that often cause contention for hardware resources or those within ASE. A classic examples include the typical

- "auditing" routines, which record which users did what operations on the associated data rows. While the auditing is important to satisfy business compliance and security issues, the loss of a small number of audit trail records in the event of a system outage is acceptable.
- "worktables" or other temporary tables which hold data only for intermediate processing or hold temporary results of mathematical models that will be inserted into persistent tables later.

Knowing this, consider the following statistics regarding the internet auditing implementation at a Sybase customer:

DBName	TableName	IndexName	IndexID	Rows Inserted	Inserts Per Sec	Rows Deleted	Rows Updated
www_db	www_access_log	www_access_log_idx3	3	547,029	19	0	0
www_db	www_access_log	www_access_log_idx2	2	547,028	19	0	0
www_db	www_access_log	www_access_log	0	547,023	19	0	0
www_db	www_access_log	www_access_log_idx4	5	547,028	19	0	0

Based on some quick checks, all this logging activity was being done in a single line of a stored procedure, which has the following usage statistics aggregated from monSysStatement:

Line Number	Num Execs	Elapsed avg	Elapsed tot	Cpu Time avg	Cpu Time tot	Wait Time avg	Wait Time max	Wait Time tot
87	33,072	25	837,852	5	183,572	19	4,300	646,808
112	33,181	0	320	0	309	0	0	0
114	33,240	0	34	0	30	0	0	0
116	2,408	0	0	0	0	0	0	0
117	2,408	0	11	0	9	0	0	0
120	30,902	0	44	0	39	0	0	0
125	30,962	0	14	0	12	0	0	0

Note that due to overrunning the statement pipe size, the above is missing a lot of the operations. However, it is obvious that line #87 is where the actual atomic insert is occurring – and equally obvious is the amount of WaitTime due to the log contention as well as the increase in CpuTime – both to perform the logical reads to process the insert as well as the log contention. While the average wait time was only 1.9ms (truncated above), at least one process was blocked due to log contention so long that it actually took 3.5 seconds of WaitTime and 950ms of CpuTime (not shown).

The speed of performing the application auditing in this procedure could be increased significantly by inserting the lines:

```
set delayed_commit on
-- line 87's insert
set delayed_commit off
```

This should dramatically drop the contention for the log semaphore as every audit record will not have to wait for the physical log write to occur. In addition, in the customer's system none of the indices were ever used. For example, between the last boot time and when the above data was collected, nearly 27 million records had been inserted in the system and the table was *never even read* from a query perspective. As a result, in addition to using delayed commit, some serious consideration should also be given to eliminating some or most of the indexes on this table as it not only incurs CpuTime to perform the LogicalReads to traverse the index to find the insertion point, it also has a high potential for contributing to "Latch Contention" if any

of the indexes are based on monotonically increasing values such as access_datetime or similar field (e.g. even an identity column such as row_id).

Separate Database

Ideally, if the above auditing tables were in a separate database, the database could simply have the 'delayed commit' option enabled via sp_dboption. The separate database in and of itself would reduce log semaphore contention. However, there are several aspects to consider in implementing an optimal solution for this.

First, consider the above example in which the auditing data is likely collected via a stored procedure (which may poll information from master..sysprocesses or system functions such as suser_name() and getdate()). The stored procedure ideally should be created in the same database as the auditing table and not in the database containing other application data. The reason why is that if the stored procedure is in one database and the table in another, it increases the number of times the DBTable spinlock has to be grabbed – which could lead to problems in high concurrency environments.

Secondly, such auditing or application logging records should not be within the scope of a transaction. Unfortunately, ASE does not implement the concept of 'autonomous transactions' in which inner nested transactions contained in an outer transaction will be allowed to commit even if the outer transaction rollsback. As a result, if you include auditing or logging information as part of the original user transaction, if the transaction fails and rollsback, you lose the auditing information that would be key evidence that someone was attempting something they shouldn't have been able to – or minimally, you lose some debugging information about when the failures occurred.

Provided that both of these tips are followed, separating auditing or application logging information into a separate database should not cause scalability issues by simply swapping the log semaphore contention for internal spinlock contention on the DBTable structure. However, a key consideration at some point should be a shift in the application architecture – especially for auditing type events. Commonly, auditing events are embedded in the application logic sent to the database from the middle-tier component using pseudo logic similar to:

```
-- first batch
insert into audit_table values (…)

-- second batch
begin tran
        DML
commit tran
```

or similar arrangements. The problem with the above is that the auditing is not only serialized with respect to the business logic (whether in the transaction or not – the connection spends time before the business transaction, during or after the business transaction recording auditing information). The degree of concurrency also increases contention on auditing data objects – whether in the form of latch contention on indexes on the audit tables or log semaphore contention on the auditing database. The best implementation for an auditing trail would be to separate the audit trail from the business logic in the middle tier using a scheme similar to the following:

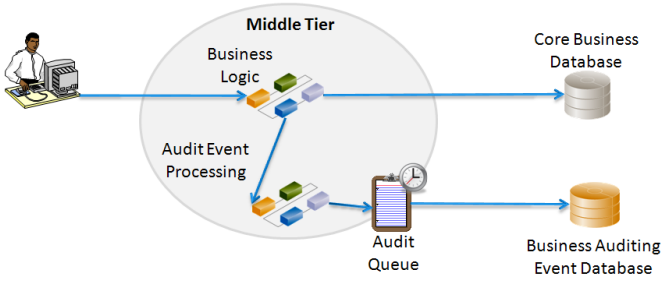


Figure 14 – Asynchronous Audit Event Processing

The above can leverage many of the techniques discussed in this section such as micro-batching the queued audit events or delayed commit.

Stored procedures with no recovery (e.g. savepoints)

A frequent problem with stored procedure implementations in high concurrency environments is the lack of transaction management within or wrapped around stored procedure executions. While this many may think this improves the procedure performance, it actually contributes to two problems:

- Reduced scalability due to log contention during concurrent procedure execution of any procedure statement that modifies data – irrespective of whether the same procedure or not.
- Recoverability issues should an error occur within the stored procedure.

The latter is interesting. Consider the auditing procedure above. The insert happens at line 87. Should the procedure fail on line 120 due to a fatal error, the question that should be asked are:

- "What should happen or will happen to the inserted row?"
- "How will the application handle the error – will it simply call the procedure again, after calling a different routine to first remove the partially inserted data?"

In a lot of cases, many stored procedures execute DML statements on multiple lines within itself or sub-procedures. While exceptions may result in errors being raised to the client application, if an error occurs in between these statements, the operations are not contained within an explicit transaction and therefore are not rolled back. Instead, the application is responsible for handling the error by removing the partially modified rows and then re-issuing the same procedure call. However each one of these atomic statements results in log contention. While it may be possible to bracket the modifications with an explicit transaction, in some cases, the statements in between cannot be within the scope of a transaction – or if they are, they could contribute to any locking contention if the locks are held longer than the individual statement.

As a result, one possibility to consider the following logic for procedures that contain multiple DML statements:

```
create procedure <procname>
…
select @delayed_commit=0
if @@trancount=0
```

```
begin
        set delayed_commit on
        select @delayed_commit=1
end
…
-- rest of stored procedure logic
…
-- just before the last DML statement
if @delayed_commit=1 set delayed_commit off
```

The result is that the overall stored procedure execution time should drop as well as a decrease in log semaphore contention. Application logic areas that could benefit from this include:

- Batch processing stored procedures that use cursors to crawl through the table to avoid escalating to table locks
- Any stored procedure that includes more than one DML statement that likely affects a small number of rows in non-tempdb tables.

Applications that Buffer Changes

Some applications buffer changes and can retain them for some time and discard them when no longer needed. A good example of this is Sybase's Replication Server, which can save already delivered DML statements in the queue for a configured save interval. When RS reconnects, it checks to see what was the last transaction delivered safely and restarts from that point. Consequently, it can safely use the delayed commit option as it will replay from the saved transactions. Later versions of RS have direct support for delayed commit built in by forcing a log flush periodically and then discarding the saved transactions. For example, every 60 seconds, it can force a log flush by turning delayed commit off, executing a row modification on a replication system table, and then committing. The transaction with delayed commit off, of course, will be done using a full blocking commit and will have to wait until the disk write completes – at which time it is then known that all previous transactions have also been flushed to disk. The saved transactions can be purged (if desired) and delayed commit re-enabled for subsequent processing.

Reduced Durability Database

As mentioned above, some data is not business critical – however, it is necessary for the business – or at least it normally near full recoverability is desired. In many systems today, extensive models are used to determine security pricing, insurance risk mitigation, or simply business forecast modeling. Whether to support these models or whether just to perform complicated calculations within the application, often a subset of worktables are used to hold the temporary data rows. This data typically is not recovered in the event of a system failure and often is repeatedly deleted and re-inserted multiple times by different users as different pricing scenarios or forecasts are run based on differing parameters.

Ideally, such tables could be contained in an IMDB – but due to the size of the data and a likely desired to restrict the amount of data cache such models use, the memory space requirements will likely exceed the amount of data cache desirable to hold the currently active models. This is where monCachedObject comes in to help by allowing DBA's to monitor the amount of cache normal and peak usage for data that is crucial and there is a desire to make sure the active rows are cached, but without over-allocating cache and causing data to displace more critical caching requirements. This is a concept that in the next chapter we will call a "restriction" cache – or

one that is of a particular size to restrict the amount of memory that otherwise might be used to just the amount necessary.

The alternative to IMDB that supports using a restriction cache is a "Reduced Durability Database" or RDDB. A RDDB is disk resident, but may or may not be recovered in the event of a failure. Consider the following table:

Durability Clause	RDDB	Recoverability Restrictions
Full (default)	No	Full recovery in the event of any failure
'at_shutdown'	Yes	transactions are durable while the server is running and after a polite shutdown. All durability is lost if the server fails and the *database is marked suspect*.
'no_recovery'	Yes	transactions are not durable to disk and all changes are lost if the server fails or is shut down. For disk-based databases, Adaptive Server periodically writes data at runtime to the disk devices, but in an uncontrolled manner. After any shutdown (polite, impolite, or server failure and restart) a database created with no_recovery is not recovered, but is *re-created from the model or template* (if defined) database.

The durability clause is an option to the create database command, such as:

```
create database <dbname>
        on <devlist>=MB
        log on <logdevlist>=MB
        with durability = { FULL | AT_SHUTDOWN | NO_RECOVERY }
```

It also can be changed by using the alter database command:

```
alter database <dbname>
        set durability = { FULL | AT_SHUTDOWN | NO_RECOVERY }
```

However, when using an RDDB, there are two critical aspects to consider:

- How RDDB will impact recoverability and ways to avoid issues
- Whether minimal logging will be able to be used to fully exploit RDDB capabilities

These aspects are discussed below, along with a section that illustrates the types of RDDBs that can be created and recommendations for using RDDBs.

Durability & Recovery

The impact of the durability clause on recovery is interesting. For RDDBs with 'NO_RECOVERY' – the answer is quite simply – the database is simply rebuilt from the template or model on each reboot no matter how the system was stopped. However, ensuring recoverability with 'AT_SHUTDOWN' takes a bit of effort.

We are all aware that if a server is shutdown "politely" (sans no wait), the ASE server does the following in preparation to halting the kernel tasks:

1. Performs checkpoint on all the databases
2. Prevents any new user from logging in
3. Waits for all running or sleeping processes to finish their job
4. Disconnects any users
5. Performs another checkpoint on the databases, this time with a flag that informs ASE that it needs to flush:

- ✓ All the dynamic thresholds in mixed log-data databases
- ✓ All the object statistics
- ✓ The values of the identity fields to avoid holes after recovery

In addition, it cleanly closes the database context. A shutdown with nowait differs significantly in that it essentially only does steps 2 & 4 and then does the shutdown. For years, a number of DBA's have implemented a slightly less impolite shutdown variant by issuing a manual checkpoint in each database before a shutdown with nowait. While this adds step 1 back into the mix, step 5 is still missing as is the necessary clean close of the database context from an RDDB perspective. Note, however, that the most common reasons for a shutdown with nowait typically are:

- Errant process than cannot be killed normally.
- Memory corruption in which a normal shutdown would result in pushing the corruption to disk.
- Attempts to do a normal shutdown appear to be significantly delayed due to user activity.

With the exception of the middle bullet, all of these provide some time to deal with the recovery issue of RDDBs and in particular, those with AT_SHUTDOWN. As noted in the table above, if there was log activity and a database with a durability of 'AT_SHUTDOWN' is shutdown with nowait or crashes, the database will not be recovered – instead it will be marked suspect:

```
00:00:00000:00001:2010/10/07 17:04:48.57 server  Recovering database 'pubs2_as'.
00:00:00000:00001:2010/10/07 17:04:48.57 server  Started estimating recovery log boundaries for
database 'pubs2_as'.
00:00:00000:00001:2010/10/07 17:04:48.59 server  Database 'pubs2_as', checkpoint=(12944, 42),
first=(12944, 42), last=(12944, 42).
00:00:00000:00001:2010/10/07 17:04:48.59 server  Completed estimating recovery log boundaries for
database 'pubs2_as'.
00:00:00000:00001:2010/10/07 17:04:48.59 server  Error: 16039, Severity: 21, State: 1
00:00:00000:00001:2010/10/07 17:04:48.59 server  The transaction log in database 'pubs2_as' shows
activity requiring recovery. The database cannot be recovered, as it has a durability level
'AT_SHUTDOWN' and will therefore be marked suspect.
00:00:00000:00001:2010/10/07 17:04:48.60 server  Error: 3414, Severity: 21, State: 1
00:00:00000:00001:2010/10/07 17:04:48.60 server  Database 'pubs2_as' (dbid 16): Recovery failed. Check
the ASE errorlog for further information as to the cause.
```

As a consequence, if you are using an RDDB, you should

- Always use a template database with durability=NO_RECOVERY
- Be prepared to drop/recreate and possibly load from dump the RDDB on restart for databases with durability=AT_SHUTDOWN.

The first bullet is obvious and doesn't result in much planning other than recognizing the additional time it may take to copy the template database for the RDDB. The second points out a rather nasty position to be in. Typically, getting a system back online for user access is a time sensitive operation – one that doesn't necessary allow time for dropping the RDDB, creating it for load, loading from a transaction log dump, and then resetting the tables to a desired restart point by truncating those containing volatile data rows, etc.

There are two ways to avoid this:

- Consider switching RDDBs with durability=AT_SHUTDOWN to FULL prior to any shutdown operation and then switching back after restart.

- Create a mountable version of the RDDB at known ‘rollback’ recovery points

The first is fairly simple, however, a good practice is to create a script with the following and ensure that all the DBAs use it instead of executing a shutdown with nowait via a direct connection.

```
alter database <dbname> set durability=FULL
go
shutdown with nowait
go
```

This works because when changing a RDDB to a DRDB by setting the durability to FULL, the first thing ASE does is flush all the pending writes for the RDDB to disk – ensuring full recoverability. For restarting, you simply add the reverse to any database reboot script that contains the dbcc tune(desbind) as well as other database server restart requirements.

The second one takes a bit more effort. Consider the following considerations with respect to restoring an RDDB:

- You can do a *full* database dump/load for an RDDB.
- Dump transaction/load transaction from an RDDB to any device is not permitted. Only dump transaction with truncate_only is permitted.
- Quiesce database from an RDDB is not guaranteed.
- Quiesce database blocks write access until hold is released.
- Quiesce database works best with databases using isolated devices – not only from a Sybase standpoint, but also in terms of physical storage – especially for raw partitions.

This discussion sort of is getting ahead of ourselves a bit as you need to realize the write blocking consideration restricts this recovery technique to more batch oriented applications. Consider the following examples:

- A daily market feed, list of insured members or other large bulk feed which overwrites existing data and then has some post-load processing associated.
- A periodic simulation model such as risk calculation or portfolio position that is recalculated on a periodic basis throughout the day.

In both of these cases, the intent would be that if a failure happened, to attempt to ‘rollback’ to the preload state or an intermediate processing state, and then restart from that point. These are ideal RDDB situations in which at key ‘rollback’ recovery positions within the processing, the controlling process would do the following:

```
-- rollback/recovery position
alter database <dbname> set durability=FULL
go
quiesce database <tagname> hold <RDDB dbname> for external dump to <manifest file>
go

-- OS process or SAN utility snapshots the device state

quiesce database <tagname> release
go
alter database <dbname> set durability=AT_SHUTDOWN
go

-- next phase of processing can continue.
```

If a failure happens, the suspect database can simply be dropped, the devices restored from the snapshot and then the database remounted. In addition to batch oriented applications, this can also be used with session oriented applications for which recovery back to a previous day's or mid-day data consistency state is permitted. A more in-depth discussion will follow later in the different types of RDDBs.

Minimal Logging

However, by itself, durability really doesn't help much to alleviate the transaction logging problem. To do so, what is needed is to enable minimal logging via:

```
create database <dbname>
        on <devlist>=MB
        log on <logdevlist>=MB
        with durability = AT_SHUTDOWN,
                dml_logging = 'minimal'
```

At a very high level, the first example would support transaction rollbacks – the second might not. What is confusing is that often when discussing relaxed durability databases, the assumption is that dml_logging was set to minimal. This is one of the key factors, but along with dml_logging, there are a few other database options to consider in creating different types of RDDBs.

Most are familiar with the concept of minimal logging thanks to fast bcp, writetext no_log, and tempdb select/into commands. Often somewhat mistakenly thought of as "no logging", what minimal logging really refers to is that the space allocations and system log records will still be recorded, but the usual data row modifications will not be. Obviously, this has an impact on recoverability (remember the dangers of fast bcp and a system crash?) but it also has an impact on transactional semantics. Minimal logging can be enabled at the database, object or session level as follows:

```
-- database level
create database <dbname>
…
with durability=at_shutdown, dml_logging={full | minimal}
go

alter database <dbname>
set dml_logging={full | minimal}
go



-- object level
create table <tablename> (
…
)
with dml_logging={full | minimal | default}
go

alter table <tablename>
  set dml_logging={full | minimal | default}

select <column_list>
  into <tablename>
       with dml_logging={full | minimal | default}
  from <table_list>
  where <predicate expressions>
go


-- session level
set dml_logging { full | minimal | default}
go
```

Note that there is one critical requirement for minimal logging – the database must have the 'select into/bulkcopy' option enabled. So creating an RDDB for minimal logging would look like the following:

```
create database <RDDB name>
        on <device list>
        log on <device list>
        with durability=at_shutdown,
                dml_logging=minimal
go

-- since recoverability isn't a consideration,…
exec sp_dboption <RDDB name>, 'trunc log on chkpt','true'
-- this one is mandatory
exec sp_dboption <RDDB name>, 'select into','true'
go
```

In the above examples, both the object level and session level settings have the added 'default' option. The reason is that the object will inherit the database's settings as will the session based on the database of the current context. This allows applications or maintenance operations to be coded without having to worry what the original setting was – rather than turning off minimal logging (which could have negative impacts if the database had minimal logging enabled), the session simply sets it back to the default. Consider:

```
-- turn on minimal logging for the archive delete crawler procedure
-- this is a simplistic example…

create procedure archive_trades @archive_date datetime
as begin
        declare @begin_row int, @end_row int, @num_rows=int, @rows_affected int
        set dml_logging minimal
        select row_id=identity(20), trade_id
          into #trades_to_archive with dml_logging=minimal
          from trade_history
          where trade_date < @archive_date
        select @rows_affected=@@rowcount

        -- pick a row count small enough to fit in the ULC
        select @begin_row=1, @end_row=1+25, @num_rows=25
        while (@rows_affected > 0)
        begin
                delete trade_history
                  from trade_history h, #trades_to_archive a
                  where h.trade_id = a.trade_id
                    and a.row_id >= @begin_row
                    and a.row_id <= @end_row

                select @rows_affected=@@rowcount

                delete #trades_to_archive
                  where row_id >= @begin_row
                    and row_id <= @end_row

                select @begin_row=@end_row, @end_row=@end_row+@num_rows
        end

        set dml_logging default
        return 0
end
```

Minimal logging can only be enabled in RDDB's – meaning that the database has durability set to 'at_shutdown' or 'no_recovery'. Currently, this means that you cannot have a minimally logged table in an otherwise fully recoverable database. Because of the implications of the durability, it also means that you cannot have a fully recoverable table in a RDDB. Think about it. If 'no_recovery', the table will be rebuilt from the template database. If 'at_shutdown', if the system experiences a crash or a shutdown with nowait, the RDDB will have to be rebuilt or restored. As a consequence, the best that can be done is that an RDDB table can be restored to a point-in-time based on a template, database dump or quiesce snapshot.

From an RDDB perspective, the minimal logging implementation largely depends on the size of the ULC – and consequently is more closely aligned with the 'session tempdb user log cache' than the bcp or select/into variants of minimal logging. As the statement is executed, the modified rows are written to the user's ULC as normal. If the transaction commits or rollsback prior to a ULC flush, the log records are simply discarded. However, if the ULC becomes full or is flushed due to another reason (change database context), then the records are written to the transaction log irrespective of the minimal log setting. As a result, often the huge deletes used by archival processes, etc. – the very commands that some have wanted to be minimally logged for a long time – will still be logged.

Note that minimal logging is not always possible. There are a number of restrictions that require full logging of modified records as listed in the documentation:

- You can use minimally logged DMLs only in in-memory or relaxed-durability databases. You cannot use them in databases that have full durability.
- The logging mode for a table in a multi-statement transaction remains unchanged throughout the entire transaction.
- All DML commands are fully logged on tables that participate in declarative or logical referential integrity constraints.
- You can export the set dml_login option from a login trigger to a client session. However, unlike most set options, you cannot export set dml_logging from stored procedures or issue execute immediate to their callers, even when you enable set export_options.
- All DML commands after a savepoint is set executes with full logging even though the table would have otherwise qualified for minimal logging.
- Full logging is performed if any active triggers exist on the table. For DML to run in minimal-logging mode, disable any triggers before executing the DML statement.
- An optimizer selecting deferred-mode update processing overrides the minimal DML logging setting, and the DML is executed with full logging.
- To support log-based replication, DML on replicated tables always performs full logging.

The highlighted points can best be summarized by the following:

- Any schema will not have DRI or triggers to enforce referential integrity – therefore the application will have to enforce this if necessary. Care must be taken when using modeling tools so that conceptual references do not get generated.
- HA/DR implementations for an RDDB will need to rely on a different mechanism than software replication.
- Explicit multi-statement transactions involving DML in an RDDB will require using a session setting of dml_logging=full, but should only be used when a rollback is specifically required. Strong consideration should be given to preferring recoverable operations.

- Cross database transactions between an RDDB and a DRDB should be split into two separate operations – the explicit transaction on the DRDB and then a non-transactional DML recoverable operation in the RDDB.
- Applications that perform large deferred deletes/updates may need to be rewritten to avoid the deferred operation if the logging impacts throughput.

Types of RDDBs

When considering the various 'create database' options, not all combinations make sense or are supported. Consider the following matrix for types of databases that can be created:

Database Type	In-Memory	Temporary	Durability= Full	Durability= no_recovery	Durability = at_shutdown	Dml_logging = full	Dml_logging = minimal	Delayed commit	Comments/Applicability
IMDB	✓			✓		✓			IMDB from template, transactional
IMDB	✓			✓			✓		IMDB from template, non-transactional
IMDB Tempdb	✓	✓		✓		✓			IMDB tempdb, transactional
IMDB Tempdb	✓	✓		✓			✓		IMDB tempdb, non-transactional
Normal (default)			✓			✓			Normal – full recoverability required
Normal DRDB			✓			✓		✓	Best effort/near full recoverability
RDDB				✓		✓			Dubious use case. Session oriented work tables, transactional
RDDB				✓			✓		Session oriented work tables, non-transactional
RDDB					✓	✓			Dubious use case. Batch oriented, transactional
RDDB					✓		✓		Batch oriented, non-transactional
RDDB Tempdb		✓		✓		✓			Dubious use case, use tempdb
RDDB Tempdb		✓		✓			✓		Dubious use case, use tempdb

For this discussion, we will focus strictly on the RDDB combinations – especially the highlighted differences with respect to durability and minimal logging. Note that in the above, the 'transactional/non-transactional' characterization refers to the aspect that minimally logged operations may not be fully rolled back and as a consequence such databases really do not support operations that require full support for rollback unless established at a session basis.

Before we discuss the two that remain, the first thing that should be noted is that while some combinations are supported, they are of dubious use cases. The easiest to consider is the notion of a RDDB tempdb. An attempt to create one would use syntax such as:

```
create temporary database rddb_tempdb
        on <dev_list>
        log on <dev_list>
        with durability='no_recovery'
go



exec sp_dboption rddb_tempdb, 'abort tran on log full','true'
exec sp_dboption rddb_tempdb, 'trunc log on chkpt','true'
exec sp_dboption rddb_tempdb, 'select into','true'
go
```

Since tempdb's are not allowed to reference a template database, there is nothing to gain in the above implementation – and everything to lose (e.g. no 'session tempdb ULC', etc.).

The obvious question for a session oriented RDDB is whether or not full transaction support is required. Remember, overall, as soon as using an RDDB, the implication is that full recoverability is not a consideration. Since full transactional support is possible within tempdb, the question that remains is why would one use an RDDB instead of a tempdb for this. The only answer is to be able to have a pre-loaded schema and/or be able to recover to a point-in-time snapshot. Remember, without minimal logging, the lack of durability is not going to necessarily contribute any scalability improvements, so it really only makes sense to consider this in the context of minimal logging…..which largely voids the transactional consideration anyhow.

Session Oriented/No_Recovery RDDB

The easiest one to start with is the RDDB tempdb as it is one of the key two generic types of RDDB's. One of the key notions of an RDDB with no_recovery that distinguishes it from a RDDB with at_shutdown is that whenever the ASE is restarted, the RDDB is reset to a known data context. This also differs from a tempdb in that in a RDDB, the schema will persist (along with any stored procedures) as well as any static or semi-static data such as reference data contained in the template database. As a result, a no_recovery RDDB is especially well suited for database that need to hold temporary work table data for individual sessions that exceeds the memory available for an IMDB. When the data reaches a persistent state, the data is copied from the work tables to the persistent database that has full recoverability. An example of this might be a electronic shopping cart for a web application. While this could use an IMDB, one consideration is that given the need to perhaps retain shopping cart items for a few days to allow users to restore interrupted internet sessions, the number of active 'shopping carts' might well exceed the amount of memory you want to reserve for this function.

Batch Oriented/At Shutdown RDDB

A session oriented RDDB is one in which the data in the tables is essentially specific to individual sessions and is likely discarded at the end of the session or when the data reaches a persistent state. In a batch oriented RDDB, the inference is that the data being manipulated is used by multiple sessions – and that the logging of all the row modifications – especially during batch processing – is impacting performance. Since the data is shared, the inference also is that the data will exist after a session exits and will also need some level of recovery to 'recovery point'. As described earlier, however, ensuring that a RDDB with AT_SHUTDOWN can be recovered quickly in a usable state requires using periodic snapshots in time of database state. If the application was more session oriented, the snapshot would only be able to taken when user activity was very minimal so that the quiesced state doesn't interfere too much with write activities.

Recommendations

The following recommendations are suggested starting points for the ASE transaction log implementation and associated configurations.

Configuration Parameter	Default	Recommendation	Tuning Considerations
user log cache size	(page size)	(2x page size)	Make sure there is a buffer pool of the same size and that the sp_logiosize setting is also identical.

Configuration Parameter	Default	Recommendation	Tuning Considerations
user log cache spinlock ratio	20	(see comments)	Reported as ULC semaphore requests/waits in sp_sysmon or WaitEventID=272. Rarely a problem and may not need to be adjusted unless exceeds 5% as with any other spinlock.
session tempdb log cache size	(page size)	(16x page size)	Considering the default 10 page temp table, this minimally should be 10 pages in size rounded up to the buffer pool size. Increase in multiples of 8x server page size (i.e. 16K for 2K server) to match large buffer pool size in cache where tempdb is bound to

To ensure that the ULC and log IO size are set correctly, the full syntax of commands is:

```
-- first make sure the buffer pool is available.  For this
-- example, we will assume a 4K server and that non-mission
-- critical databases use the default data cache for their log

-- verify absence or presence of 8K pool in default data cache
-- we are using 8K as that is 2x the server page size

exec sp_poolconfig 'default data cache'
go

-- if 8K pools is not present, add one – since this is a 4K
-- server, we will need to steal space from the 4K pool most
-- likely, although if a 8x pool (32K) is configured and likely
-- over configured, we might instead steal from there

exec sp_poolconfig 'default data cache', '100M', '8K', '4K'
go

-- now we need to make sure the databases will use the pool
-- as their default logiosize was reset to the server pagesize
-- since the pool was missing:

exec sp_logiosize <dbname_1>, '8K'
exec sp_logiosize <dbname_2>, '8K'
…
exec sp_logiosize <dbname_n>, '8K'
go
```

When considering log contention, the following rules should be considered:

Feature or Technique	Consider	Don't Consider
Async Log Service	When ***ALL*** of the following are true: • full durability is required • log semaphore contention >30% • 16-20 engines • log device speed ~1ms/IO or less • log semaphore waits >10ms • small OLTP transactions • >1 write per log page	When any of the following apply: • full durability is not required • <12 engines • log speed/waits higher than semaphore waits • ALS waits are greater than 10ms • large transactions or <1 log write per page
Delayed Commit	Consider when any of the following are true • minor transaction loss in the event of a system failure is permitted • stored procedure is not executed within a transaction nor explicitly uses a transaction but has multiple non-tempdb DML statements or a non-tempdb DML statement in a loop • batch processing outside the scope of a transaction • applications that buffer, detect and automatically replay lost transactions • High speed atomic transactions with forced log flush or >1 write per log page	When any of the following are true: • Recovery after normal shutdown not necessary (use RDDB) • Full durability is required • Transaction size normally is 1 or more log pages (or fills the ULC) (likely won't benefit)
Micro-Batching	Consider when all of the following are true: • Application is almost exclusively inserts • Single row/txn response time allows buffering in application to implement batching • Only one or a few tables are involved	When any of the following are true: • Primarily a GUI/user driven application • Mostly queries with few DML statements • Batch processing using bulk DML statements with predominately updates or deletes (e.g. an

Feature or Technique	Consider	Don't Consider
		archive process)
Reduced Durability Database	Consider when all of the following are true: • Data recovery after an abnormal shutdown is not required (e.g. worktables) or recovery allows a 'restore point' • Schema persistence in the event of an abnormal shutdown is required • Transaction sizes mostly fit within the ULC size • Data volume exceeds available memory (e.g. IMDB is not an option)	When any of the following are true: • Transactions regularly are larger than the ULC size (e.g. large batch processes such as archive) • Full or nearly full durability is required in the event of an abnormal shutdown • significant percentage of DML statements are not minimally logged compliant (e.g. tables with triggers, mixed transaction modes, replicated tables, etc.) • schema persistence across shutdowns is not necessary (use an separate tempdb instead)

Named Caches & Procedure Cache

In the mid-1990's, Sybase became aware that as the number of user processes increased and the data volumes increased in OLTP systems, that a single flat memory model simply doesn't scale. It is only common sense that:

- Some data is written once, rarely read – and therefore does not need a long MRU→LRU chain to facilitate caching.
- Large volumes of data read from a single table can cause other critical tables to be flushed from cache.
- Highly critical processing tables needed to be protected.
- In order to facilitate large I/O's – especially for reads, larger contiguous buffer pools need to be available
- With multiple users all attempting concurrent I/O's, the notion of grabbing from a single MRU→LRU buffer chain leads to contention at the edges of the chain – both in multiple users trying to grab the LRU buffer as well as appending to the front of the MRU chain. The latter happens frequently as pages that are accessed before they reach the LRU are moved back to the front of the chain.

In order to provide scalability, a DBMS system must provide the following facilities:

- The ability to separate items in cache – restricting some from using too much cache, while protecting others from cache volatility.
- Multiple sizes of buffer pools for different I/O sizes.
- The ability to partition caches into multiple MRU→LRU chains

Of course, each of these aspects need to be tunable. This section describes some configuration advice and tuning considerations for using named caches.

Cache Structure

We are all familiar with the common MRU→LRU page chains implemented in most caching systems. The problem is that there are two big contention points with such implementations that impact scalability:

- Contention for clean pages from the LRU
- Contention for appending recently used pages to the MRU

While the second one may appear to be due to the first, remember, that a page used 5 minutes ago may not have even reached the wash area yet – especially if the cache is very large. As a result, any maintenance done within the MRU→LRU chain to relink the pages is viewed the same and needs the spinlock. As per the documentation:

> *As the number of engines and tasks running on an SMP system increases, contention for the spinlock on the data cache can also increase. Any time a task needs to access the cache to find a page in cache or to relink a page on the LRU/MRU chain, it holds the cache spinlock to prevent other tasks from modifying the cache at the same time.*
>
> *With multiple engines and users, tasks must wait for access to the cache. Adding cache partitions separates the cache into partitions, each of which is protected by its own spinlock. When a page needs to be read into*

cache or located, a hash function is applied to the database ID and page ID to identify the partition that holds the page.

-- Adaptive Server Enterprise 15.0.2 Performance and Tuning Series: Basics; Chapter 5 - Memory Use and Performance; Configuring the data cache to improve performance

A data cache in ASE is a linked list of buffers that contain the logical pages from the various databases. Each cache has a cache lookup hashtable (this is part of the cache overhead) to track which pages are in cache and the address of each page in memory. Each buffer is a pool sized area of memory that contains 1 or more logical datapages as illustrated below:

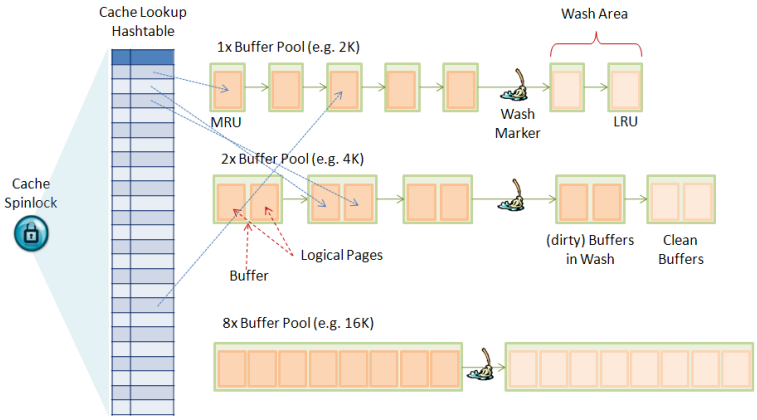


Figure 15 – Cache Spinlocks, Buffer Pools & MRU→LRU Chains

Each named cache then has:

Cache Hashtable – A list of database pages contained in the cache. This list is crucial to the speed of finding pages in cache vs. serially scanning the cache contents. The cache hashtable is part of the cache overhead when creating a new cache or resizing an existing cache.

Buffers – A logical structure containing the MRU→LRU pointers, the MASS bit, and other internal structures that protect the data pages in cache. Buffers can be thought of as a wrapper around the logical database pages in 1x, 2x, 4x and 8x the server page size. The buffer structures are also part of the cache overhead.

Buffer Pools – Groups of 1x, 2x, 4x and 8x server page size buffers containing logical database pages accessed through large I/O operations.

Memory Address Space Segment (MASS) – A continuous chunk of memory that contains all the pages within a single buffer. While a buffer really is a series address pointers, if thought of as a wrapper around pages, then the MASS can simply be thought of as the collection of pages in the buffer. For example, in a 8x buffer, the 8 logical database pages are the "MASS"

Cache Spinlock – A mutex that controls multi-user access during changes to the MRU→LRU chain as well as changes to the hashtable contents.

All read/write I/O operations occur at the MASS or buffer size level – whether a 1x or 8x buffer. As a result, all the pages within a single buffer are contiguous on disk as well as belonging to the same extent. This works as follows:

- The server determines the size of I/O to be used.

- The dbid and logical pageid are used as hashkeys to do a buffer search in the hash table. Since the dbid and logical pageid are unique, this provides a direct mapping to a single buffer.
- If the page is not found, the buffer from the LRU of the requested I/O size pool is grabbed, the hash table updated (with cache request) and the physical IO issued.
- If the page is found, the buffer is relinked to the front of the MRU→LRU chain and the cache management functions end.
- If the page was not found, once the IO returns, the buffer is linked into the MRU and the hash table is updated with the cache status.

Consider the follow I/O oriented WaitEvents:

ID	WaitEvent Description	Most Common Cause
29	waiting for regular buffer read to complete	Physical read from disk. These are page-sized reads (1x buffers). See Event #124.
30	wait to write MASS while MASS is changing	Checkpoint or HK is waiting to flush a buffer to disk but needs to wait on another SPID that is currently modifying a page in one of the buffers
31	waiting for buf write to complete before writing	HK , checkpoint or wash induced IO is waiting for another SPID to finish a logical write before flushing to disk. This is the opposite of 52.
32	waiting for an APF buffer read to complete	
33	waiting for buffer read to complete	Waiting for a logical read – possibly because cache spinlock is held or timeslice expired.
34	waiting for buffer write to complete	Waiting to modify a page in memory – possibly because the cache spinlock is held
35	waiting for buffer validation to complete	
36	waiting for MASS to finish writing before changing	A SPID is waiting to make a change to one of the pages in the buffer, but first has to wait for the checkpoint, HK or another SPID that is flushing the buffer to disk.
37	wait for MASS to finish changing before changing	A SPID is waiting to make changes to a page header; but must wait until another spid finishes its changes
38	wait for mass to be validated or finish changing	
46	wait for buf write to finish getting buf from LRU	MRU→LRU relinking due to rows being added to a table causing a new page to be allocated – such as an insert/select; select/into or an update that causes a row expansion (page split or row forwarding). LRU page may be dirty (cache stall) hence the wait.
47	wait for buf read to finish getting buf from LRU	Cache stall when attempting to read a new page into memory.
51	waiting for last i/o on MASS to complete	Current SPID data modification waiting due to blocking IO on last statement (such as index tree maintenance/rebalance)
52	waiting for i/o on MASS initiated by another task	Current SPID data modification waiting for physical write initiated by another task (checkpoint, housekeeper, etc.)
124	wait for mass read to finish when getting page	Waiting on a large I/O read, most likely an APF
125	wait for read to finish when getting cached page	Waiting on a physical read issued by another SPID for the same page but not yet in cache.

The reason that some WaitEvents can be difficult to interpret are simply due to fact they refer to the internal structures and need to be put in context. For example, the implication of WaitEventID=125 is that one task needs to read a page which a second task has added to the hash table (so it will be in cache) but the physical read has not yet returned. Just thinking about

it, if adding a new page, it doesn't make sense to grab the spinlock on the cache and hold it until the read returns to place the page on the MRU. Instead, the process grabs the spinlock once to update the hash table with the dbid+pageid and then latter again once the IO completes it grabs it again to update the LRU→MRU linkage. If the spinlock was held the entire time, IO concurrency would really suffer. Of course, the problem is that another process can also ask for the same page after it has been requested but before it gets read – which is what event 125 is illustrating. On the other hand, others are difficult to interpret as there can be many different causes. For example, WaitEventID=33 could have several possibilities such as one task that wants to modify a page (LogicalRead to find the insertion point), but must wait as another task is currently reading the page – and relinking the buffer to the front of the MRU chain. While all the tasks perform these operations, the process may not always sleep on the event, consequently, it may not appear in monProcessWaits.

Every cache operation takes place at the buffer level – such as MRU→LRU placement as well as physical IO (remember the MASS bit for writes?). When a page is read into a buffer, the hashtable is updated and the buffer with the page is located at front of the MRU chain. Subsequent reads or writes simply relocate the buffer to the head of the MRU chain. The manipulation of ASE's cache is controlled by a spinlock for each MRU→LRU page chain in cache. Consider the following sp_sysmon fragment:

```
Cache: default data cache
                           per sec      per xact       count  % of total
-------------------------  ------------  ------------  ----------  ----------
  Spinlock Contention              n/a           n/a         n/a       1.0 %

  Utilization                      n/a           n/a         n/a      45.6 %

  Cache Searches
    Cache Hits                459456.7         175.5   830238191      99.9 %
       Found in Wash            9993.1           3.8    18057476       2.2 %
    Cache Misses                 466.5           0.2      843002       0.1 %
-------------------------  ------------  ------------  ----------
  Total Cache Searches        459923.2         175.7   831081193

  Pool Turnover
    2  Kb Pool
       LRU Buffer Grab          8381.3           3.2    15145013      86.3 %
         Grabbed Dirty             0.0           0.0           0       0.0 %
    4  Kb Pool
       LRU Buffer Grab           227.6           0.1      411270       2.3 %
         Grabbed Dirty             0.0           0.0           0       0.0 %
    16 Kb Pool
       LRU Buffer Grab           255.8           0.1      462164       2.6 %
         Grabbed Dirty             0.0           0.0           0       0.0 %
-------------------------  ------------  ------------  ----------
  Total Cache Turnover           9706.9           3.7    17540336
```

As illustrated earlier, each named cache has at least one MRU→LRU page chain per buffer pool. Highlighted in the above is the number of LRU buffer grabs (contention point #1 from above) as well as the overall spinlock contention. In the above case, a 1% spinlock contention isn't too bad, but needs to be watched. However, this is an average across all the spinlocks. Consider the following sp_sysmon fragments and cache spinlocks output from the sp_sysmon_spinlock procedure (Appendix B):

```
Cache: default data cache
                           per sec      per xact       count  % of total
-------------------------  ------------  ------------  ----------  ----------
  Spinlock Contention              n/a           n/a         n/a       1.9 %

  Utilization                      n/a           n/a         n/a      46.6 %

  Cache Searches
    Cache Hits                444812.4        2550.9    14233997      99.9 %
       Found in Wash           30218.0         173.3      966976       6.8 %
    Cache Misses                 361.3           2.1       11562       0.1 %
-------------------------  ------------  ------------  ----------
```

```
    Total Cache Searches          445173.7        2553.0    14245559


  Cache: systables_cache
                                       per sec      per xact       count  % of total
  -------------------------  ------------  ------------  ----------  ----------
    Spinlock Contention                 n/a           n/a         n/a       0.5 %

    Utilization                         n/a           n/a         n/a      15.3 %

    Cache Searches
      Cache Hits                 146080.4         837.7     4674573     100.0 %
         Found in Wash             1668.5           9.6       53393       1.1 %
      Cache Misses                    0.0           0.0           0       0.0 %
  -------------------------  ------------  ------------  ----------
    Total Cache Searches         146080.4         837.7     4674573


  Cache: tempdb_cache
                                       per sec      per xact       count  % of total
  -------------------------  ------------  ------------  ----------  ----------
    Spinlock Contention                 n/a           n/a         n/a       1.5 %

    Utilization                         n/a           n/a         n/a      38.2 %

    Cache Searches
      Cache Hits                 360397.8        2066.8    11532731      98.7 %
         Found in Wash             5112.2          29.3      163590       1.4 %
      Cache Misses                 4680.7          26.8      149782       1.3 %
  -------------------------  ------------  ------------  ----------
    Total Cache Searches         365078.5        2093.6    11682513


Spinlock Activity Report
------------------------

Spinlock Waits                                 per sec    per xact       count  contention
-----------------------------------     ----------  ----------  ----------  ----------
  default data cache::47                     3350.9        19.2      107230       6.4 %
  systables_cache::46                        3179.1        18.2      101732       6.4 %
  default data cache::55                     2162.1        12.4       69186       5.0 %
  default data cache::54                     4626.3        26.5      148043       4.0 %
  default data cache::58                     1326.7         7.6       42453       3.6 %
  default data cache::50                      982.5         5.6       31439       3.4 %
  tempdb_cache::40                            593.9         3.4       19004       3.4 %
  default data cache::28                      726.3         4.2       23240       3.0 %
  systables_cache::62                        1117.6         6.4       35764       2.9 %
  tempdb_cache::51                            565.5         3.2       18096       2.7 %
  tempdb_cache::32                            541.8         3.1       17337       2.7 %
  default data cache::27                      474.8         2.7       15194       2.7 %
  default data cache::3                       593.6         3.4       18994       2.6 %
  default data cache::26                      470.5         2.7       15056       2.5 %
  default data cache::21                      523.6         3.0       16754       2.4 %
  default data cache::45                      402.2         2.3       12869       2.3 %
  default data cache::18                      507.1         2.9       16228       2.3 %
  default data cache::17                      361.4         2.1       11564       2.2 %
  tempdb_cache::33                            448.7         2.6       14358       2.2 %
  default data cache::41                      456.0         2.6       14593       2.2 %
  default data cache::42                      421.7         2.4       13495       2.1 %
  default data cache::43                      423.9         2.4       13565       2.1 %
  default data cache::20                      452.6         2.6       14483       2.1 %
  default data cache::46                      326.9         1.9       10460       2.1 %
  default data cache::56                      306.3         1.8        9800       2.0 %
  default data cache::8                       497.7         2.9       15925       1.9 %
  default data cache::22                      294.1         1.7        9410       1.9 %
  tempdb_cache::42                            285.3         1.6        9128       1.9 %
  default data cache::44                      229.4         1.3        7342       1.8 %
  tempdb_cache::36                            352.0         2.0       11264       1.8 %
  tempdb_cache::45                            312.6         1.8       10003       1.8 %
  default data cache::36                      316.6         1.8       10130       1.8 %
  default data cache::49                      285.8         1.6        9146       1.7 %
  default data cache::1                       227.4         1.3        7277       1.7 %
  default data cache::57                      182.6         1.0        5843       1.7 %
  default data cache::2                       251.5         1.4        8048       1.6 %
  default data cache::37                      215.5         1.2        6896       1.6 %
  default data cache::39                      267.3         1.5        8552       1.6 %
  tempdb_cache::18                            267.7         1.5        8566       1.6 %
  default data cache::9                       201.8         1.2        6458       1.6 %
  default data cache::52                      179.1         1.0        5732       1.6 %
  default data cache::0                       201.7         1.2        6455       1.6 %
  tempdb_cache::5                             238.4         1.4        7629       1.6 %
…
```

How can sp_sysmon report only 1.5% contention for the 'default data cache' when we can see as high as 6.4%??? The answer is that this was from a 64 engine ASE that had cache partitioning set to 64 as well (the '::##' behind each cache represents one of the cache partitions). So, not only does sp_sysmon average out the spinlock contention over the sampling time, it also averages it out across the cache partitions.

There are three ways to reduce cache spinlock contention:

- Increase the number of cache partitions
- Change the cache replacement policy
- Bind the objects to different named caches

These along with a discussion on tuning the cache asynchronous prefetch limit and cache strategy are discussed below.

Cache Partitions

Partitioning a cache splits a cache into multiple equally sized cachelets. As documented, the number of partitions must be an even power of 2: 1, 2, 4, 8, …64 – with the maximum of 64 partitions.

Buffer Pools & Wash Markers

The first thing to remember is that each buffer pool in each cachelet must have a minimum of 256 pages. Consequently, the more a cache is partitioned, the more memory needed for each buffer pool. Consider the following table, which lists the amount of memory needed per buffer pool based on the server page size when partitioned.

Partitions	2K	4K	8K	16K
1	0.5	1	2	4
2	1	2	4	8
4	2	4	8	16
8	4	8	16	32
16	8	16	32	64
32	16	32	64	128
64	32	64	128	256

Consequently, each buffer pool would need to have a minimum of 32MB for a 2K server. This can have an impact when creating really small caches with multiple buffer pools.

By default, the wash marker is placed at 20% of the pool size for pools less than 300MB and at 60MB for pools greater than 300MB (20% of 300 = 60). However, the minimum size for a wash area is 10 buffers. A common situation is a DBA will configure a fairly small cache for tempdb – for example, a 500MB cache with 50MB 4K pool for the log, 100MB in a 16K pool and the rest (350MB) in the 2K pool. Consider the following table illustrating the number of cache partitions and the wash marker locations (wash area size) for each of these pools (size is for each cachelet in form n+m where m is the wash area size).

Partitions	350MB 2K	50MB 4K	100MB 16K	Comments
1	290+60MB	40+10MB	70+20MB	
2	140+35MB	20+5MB	40+10MB	

Partitions	350MB 2K	50MB 4K	100MB 16K	Comments
4	70+17.5MB	10+2.5MB	20+5MB	
8	35+8.7MB	5+1.2MB	10+2.5MB	
16	17.5+4.3MB	2.5+0.6MB	5+1.2MB	
32	8.7+2.2MB	1.2+0.3MB	2.5+0.6MB	
64	4.3+1.1MB	640+160KB	1.2+0.3MB	
	20KB	40KB	160KB	Minimum wash area size

Remember, however, that it is a hash on the dbid and pageid that determine which partition a page is placed into. Consequently (for example), a really large select/into in a dedicated named cache for tempdb would be scattered among the partitions vs. overflowing a single partition.

The hash table is used to map the database logical page to the buffer that it is using in cache. When a cache is partitioned, the cache is simply divided into n even chunks, ranged on the hash keys, with an even number of buffers split between the caches as illustrated below:

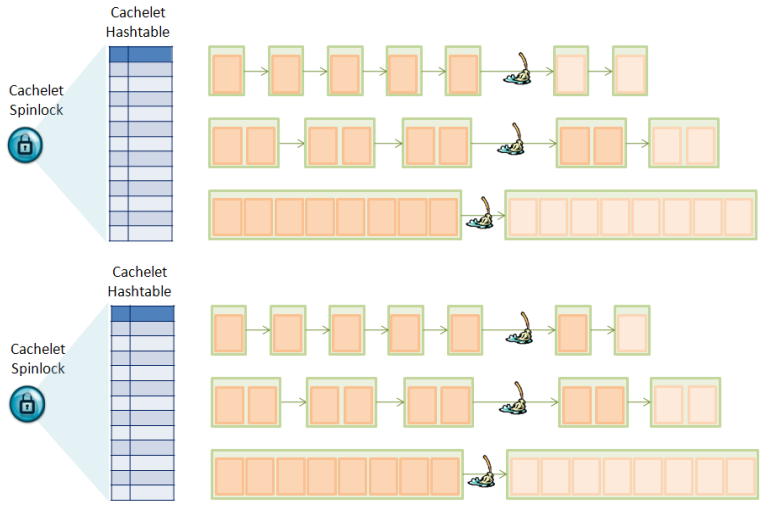


Figure 16 – Sample Cache Partitioning for Cache Partitions=2

Once it is partitioned, concurrent tasks on different engines are possibly working on different cachelets and therefore hitting different spinlocks. This assumes that access to the pages is well distributed between different users/engines. Consider the following:

- When ASE needs to do a logical read on a page (dbid+pageid) because of the partitioning, it knows which one of the cachelets the page should belong to
- If the page is in cache, then the page is simply relinked to the MRU chain in that cachelet – consequently only that spinlock is grabbed by the SPID
- If the page is not in cache, then it will be physically read from disk and added to the MRU chain for that cachelet (since the page hash values have already determined the cachelet to which it belongs). Again, only that cachelet's spinlock will be grabbed.

The above is for a single page – and very few queries ever involve just a single page as an index traversal will immediately add several pages to even a single row query. Remember, however, that a process is only performing a logical read on one page at a time, consequently, there is no danger of 'cache spinlock deadlocks' in which one process is reading one page (and holding the

spinlock) while attempting to read a second page (and waiting for the spinlock) while another process is doing the opposite.

While the concept of partitioning a cache sounds immediately like a great idea, consider the following negative aspects:

- Scanning the cache – whether a range scan, covered index scan, or a table scan – will require accessing multiple cachelets and possibly grabbing/releasing multiple spinlocks – some more than once.
- Cache utilization may be less than allocated amount.
- A small buffer pool may suffer from being "washed out".

Cache Utilization & Cache Partitioning

This takes a bit of explaining. Remember, it is the dbid + pageid that determine the hash table's lookup value, and that all the pages in the same extent having the same hashkey value. While the latter aspect may seem tough to achieve, since all extents within ASE are allocated on even 8 page boundaries (and to the same object), given any page number, it would be simple to determine the first page in the extent – and that pageid could be the pageid used for the hash function.

With a single partition (or unpartitioned) cache, all the pages in a cache can be used. However, if a cache is partitioned, unless the pages in cache's hash values are evenly distributed across all the partitions, there could be an imbalance where one cachelet will have more pages in use than others. This can quite easily happen since a page can only belong to a particular object and that object may be bound to a different named cache. For example, in databases that contain text/image data, the predominant allocation of pages frequently is for the text/image data chains – which are typically bound to a separate named cache. As a result, there are much fewer pageid values in use by data in the default data cache. Consequently, the hashing function employs some intelligence to attempt to evenly distribute the pages vs. a standard page mod() function or similar simplistic algorithm - but still could result in an imbalance.

The degree to which an imbalance could happen depends on the size of the cache relative to the sum of the database sizes within the server instance. Consider the following:

- A 20GB cache with 100% in 2K pool would have a cache overhead of ~1,280 for the hash table and buffer overhead.
- This leaves 19.2GB of cache space for a total of 9,830,400 pages that can be cached. The wash size will be at 60MB or 30,720 pages (see discussion on "wash out" to see how this is determined).
- As a result, the cache will have a pool of 9,830,400 buffers

Let's ignore the scenario of multiple databases for now. Let's also assume that we have a 100GB database – which means we could cache approximately 20% of it. Now, the logical pageid restarts for each database and is sequential starting from zero. This is evident if you run the following query in master:

```
1> select dbid, segmap, lstart, size, vstart, vdevno, next_lstart=lstart+size
2>   from master..sysusages
3>   order by dbid, lstart
4> go
```

dbid	segmap	lstart	size	vstart	vdevno	next_lstart
1	7	0	10240	4	0	10240
2	7	**0**	**1536**	23556	0	**1536**
2	7	**1536**	**25600**	0	2	**27136**
2	7	**27136**	**25600**	0	5	**52736**
2	7	**52736**	**25600**	0	6	**78336**
2	7	**78336**	**25600**	51200	2	**103936**
2	7	**103936**	25600	51200	5	129536
2	7	129536	25600	51200	6	155136
2	7	155136	25600	102400	2	180736
2	7	180736	25600	102400	5	206336
2	7	206336	25600	102400	6	231936
2	7	231936	25600	153600	2	257536
2	7	257536	25600	153600	5	283136
2	7	283136	25600	153600	6	308736
2	7	308736	25600	204800	2	334336
2	7	334336	28672	204800	5	363008
2	7	363008	28672	204800	6	391680
2	7	391680	3072	256000	2	394752
3	7	0	1536	20484	0	1536
4	3	0	76800	0	7	76800
4	4	76800	64000	0	8	140800
5	3	0	128000	256000	17	128000
5	4	128000	64000	0	18	192000
6	3	**0**	**102400**	307200	7	**102400**
6	4	**102400**	**64000**	256000	8	**166400**
6	3	**166400**	76800	153600	7	243200
6	3	243200	128000	0	17	371200
6	4	371200	64000	128000	8	435200
7	3	0	25600	0	9	25600
7	4	25600	12800	0	10	38400
7	3	38400	51200	51200	9	89600
7	4	89600	25600	25600	10	115200
8	7	0	128000	0	11	128000
8	7	128000	128000	0	12	256000
8	7	256000	128000	0	13	384000
8	7	384000	64000	0	14	448000
9	3	0	524288	0	15	524288
9	4	524288	131072	0	16	655360
31513	7	0	1536	26628	0	1536
31513	7	1536	10240	0	4	11776
31514	7	0	51200	0	1	51200
31515	7	0	25600	0	3	25600

(42 rows affected)

While the vstart is a computed value based on the vdevno and server page size, lstart (logical page id starting number) is a computed value for each dbid and size of each chunk. Notice in the above, that with each change in dbid, the first lstart value is 0. Also, that for each database (as the two highlighted show), the lstart is sequential for the chunk if you include the size. So a 100GB database will have 52,428,800 2K pages starting at lstart=0 and ending with logical pageid=52428799. Let's also assume that we have a hash key and hash chain like the locking hash tables, but that the number of hashkey buckets are fairly high so that finding a hash value is extremely fast. For example, with 9.8 million buffers, we might consider a hash table of 3 million hash buckets – each with a max of 3 dbid+pageid pairs of pages in cache along with the buffer address. Remember, the only value of the hashkey is in determining if the page is already in cache, so the exact formula for determining hashkey values is irrelevant.

However, with partitioning, we have split those hashkey values into ranges. If we partitioned the above cache into 16, we would have 614,400 buffers and hash values per partition. Simply due to the fact that some pages will never be read or the objects are bound to different caches, there may be some hashkey values that are never hit – or have fewer entries than others. Knowing this and that the hashkeys are range partitioned, we know it is then possible that one of the cachelets will be much more active. The hash table isn't the issue here – it is actually the association of the buffers. We really only have two choices:

- either the buffers are divided equally …

- …or, the buffers are members of a global pool and associated as needed to the cachelets

In the case of the latter, each new physical read would require the task to grab two spinlocks – one being from where the buffer was being grabbed from (either a global pool or another cachelet); and the other the spinlock on the MRU→LRU chain in the cachelet to which the page read is happening. If a global pool was used, this would result in spinlock contention on the global pool – defeating the purpose of cache partitioning. If the buffer was grabbed from any other cachelet, then we could have 'cache spinlock deadlocks'. On the other hand, if the buffers are divided equally, than with the imbalance of the hashkeys, there will be a similar imbalance in the cache utilization. Consequently, ASE divides the buffers evenly among the cache partitions.

If the buffers are divided evenly, you might wonder how a hash function could return a range of values that would allow a fairly even distribution of buffers. For example, with at least 5 or more databases (including system databases such as master, model, sybsystemprocs, tempdb at a minimum) each with a maximum logical page count of 2 billion, the possible range of values if hashed uniquely would be between 0 and 8 billion values. If you hashed them evenly, since master, model and sybsystemprocs are fairly small, many of the larger logical page id's for those values would never be reached. However, ASE could do a ranged hash key simply by using a modulo of the number of buffers in the named cache as the final hash key value. Consider the following simplistic example (remember our earlier example of 9,830,400 buffers in 16 partitions for 614,400 buffers per partition):

```
-- example logic of how to range partition hashkey values

1> declare       @pagenum                 unsigned int,
2>               @num_buffers             unsigned int,
3>               @buffers_per_partn       unsigned int
4> select @pagenum=0, @num_buffers=9830400, @buffers_per_partn=614400
5> while @pagenum<=10
6> begin
7>       select page_id=@pagenum, hash_key=hashbytes('ptn',@pagenum),
8>               ranged_hash_key=hashbytes('ptn',@pagenum)%@num_buffers,
9>               partn_num=(hashbytes('ptn',@pagenum)%@num_buffers)/@buffers_per_partn
10>      select @pagenum=@pagenum+1
11> end
12> go

 page_id    hash_key   ranged_hash_key partn_num
 ---------- ---------- --------------- ----------
          0 3207504971         2794571          4
          1 3738947776         3395776          5
          2 1665638974         4301374          7
          3 2732715400         9694600         15
          4 2178382307         5863907          9
          5 1067249339         5566139          9
          6 3467494846         7194046         11
          7  597376728         7552728         12
          8 3756686780         1473980          2
          9 2473996676         6566276         10
         10 3792428981         7724981         12
```

The above is just a simplistic example, but you can see even it that the pages may not be quite evenly distributed between the partitions.

The problem is that it is not easy to determine how much of a cache is not being used due to the imbalance. Consider the following example snippet from a sp_sysmon output:

```
Data Cache Management
---------------------

  Cache Statistics Summary (All Caches)
  -------------------------------------
```

```
                                  per sec      per xact       count  % of total
                             ------------  ------------  ----------  ----------

    Cache Search Summary
      Total Cache Hits          951290.7        5455.4    30441301       99.5 %
      Total Cache Misses          5042.0          28.9      161344        0.5 %
  -------------------------  ------------  ------------  ----------
    Total Cache Searches        956332.7        5484.3    30602645

…

------------------------------------------------------------------------------
  Cache: default data cache
                                  per sec      per xact       count  % of total
  -------------------------  ------------  ------------  ----------  ----------
    Spinlock Contention               n/a           n/a         n/a        1.9 %

    Utilization                       n/a           n/a         n/a       46.6 %

    Cache Searches
      Cache Hits                444812.4        2550.9    14233997       99.9 %
         Found in Wash           30218.0         173.3      966976        6.8 %
      Cache Misses                 361.3           2.1       11562        0.1 %
  -------------------------  ------------  ------------  ----------
    Total Cache Searches        445173.7        2553.0    14245559
…
```

It might be tempting to think that because the default data cache is only showing 46.6% utilization and that nearly 7% of the pages were found in wash that the cache partitioning has resulted in very low utilization such that a few cache partitions are very active. However, utilization in sp_sysmon does not mean how much of the cache was actually in use but rather how many of the cache searches hit a specific cache. This is documented in the ASE Performance & Tuning docs on sp_sysmon with the paragraph:

> *"Utilization" reports the percentage of searches using this cache as a percentage of searches across all caches. You can compare this value for each cache to determine if there are caches that are over- or under-utilized. If you decide that a cache is not well utilized, you can:*
>
> - *Change the cache bindings to balance utilization. For more information, see Chapter 5, "Memory Use and Performance," in Performance and Tuning: Basics.*
> - *Resize the cache to correspond more appropriately to its utilization.*
>
> *-- Adaptive Server Enterprise 15.0.2 Performance and Tuning Series: Monitoring Adaptive Server with sp_sysmon; Chapter 2 Monitoring Performance with sp_sysmon; Data Cache Management; Cache Management by Cache; Utilization*

Consequently, if you take the total cache searches in all caches of 30,602,645, the cache searches that occurred in the default data cache were 14,245,559 – which is 0.46550091 or 46.6%.

To find out how much of a cache actually is active, you need to query the MDA tables using a query like:

```
select c.CacheName, o.CacheID, sum(o.TotalSizeKB)/1024
 from monCachedObject o, monDataCache c
 where c.CacheID=o.CacheID
 group by c.CacheName, o.CacheID
```

An alternative is to use the sp_sysmon Cache Wizard – which uses the same MDA tables to derive its information.

Cache Replacement Policy

Another way to decrease the spinlock contention is to change the cache replacement strategy. This is especially effective for caches that have a 95% cache hit rate. The key to this is remembering that the spinlock for a cache is normally held for two reasons:

- when changing the MRU→LRU page chain

- when modifying the hashtable due to adding or removing a page from cache

If the MRU→LRU page chain is not altered, then the spinlock duration is not nearly as long and therefore contention drops. Further, if cache turnover is very low due to high cache hit ratios, then the second reason for the spinlock to be held is also avoided.

The way this is achieved in ASE is by using a 'relaxed' cache replacement strategy. In the typical named cache, the cache replacement is a 'strict' LRU replacement in which as illustrated earlier, the new buffers to hold a newly cached page are grabbed from the LRU end of the MRU→LRU chain. In a 'relaxed' cache replacement strategy, it simply is the next 'victim' in the cache.

To understand how this works, consider the normal MRU→LRU chain, but instead of terminating at the LRU end, the LRU page is linked to the MRU forming a circular cache instead of singular. Since the pages will not be relinked and consequently no movement within the MRU→LRU chain, instead of the pages 'hitting' the wash marker due to relinking, instead the wash marker moves around the cache in circular fashion as illustrated below:

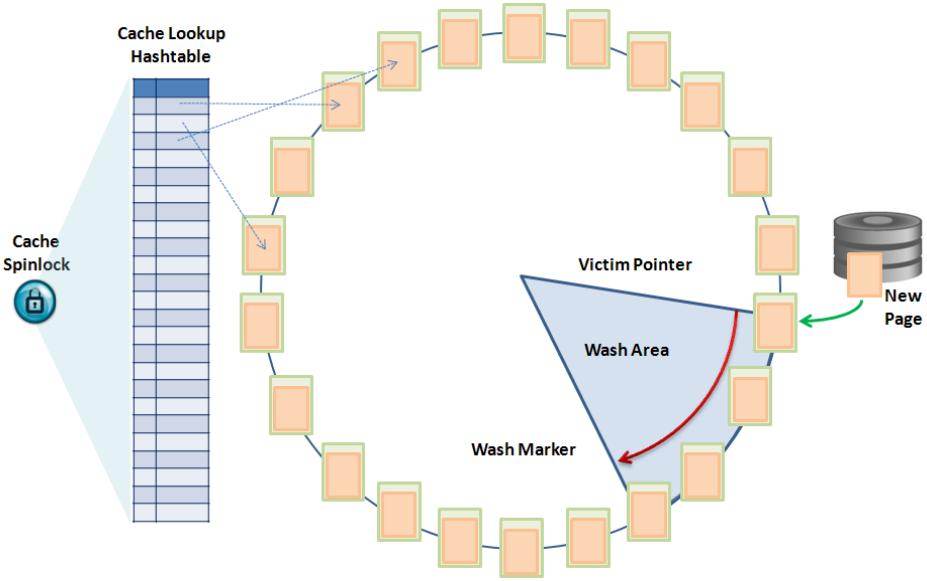


Figure 17 – Relaxed LRU Cache Replacement Policy and Wash Area

The way this works is as follows:

- Whenever a page is used, instead of relinking the page to the MRU, the buffer's 'recently used' bit is set.
- When a page needs to be read in from disk, the new page is added at the end of the wash area called the 'victim pointer' – essentially the LRU
- When a page is added, the entire wash area moves to the next page – causing the wash marker and victim pointer to be pointing to new pages.
- If the 'recently used' bit is set for the page the victim pointer is pointing to, the wash area moves to the next page until the 'recently used' bit is not set.
- As the wash area moves, when the wash marker hits a page, it clears the 'recently used' bit and if the page is dirty, initiates I/O on the page.

- The housekeeper task does not process caches with a relaxed cache strategy, consequently modified pages are only flushed to disk by the checkpoint process or wash marker.

At first this may seem like the victim pointer should never hit a page with the 'recently used' bit set since the wash marker is clearing the bit. However, if a page is 'found in wash' during a normal cache search, the 'recently used' bit would be reset to 1. That way pages that were most recently used will stay in cache until the next time the wash marker reaches them. A curious aspect of this strategy is that if the cache is static, the wash marker will never move. The documentation states that:

> *Use the relaxed replacement policy only when both of these conditions are true:*
>
> - *There is little or no replacement of buffers in the cache.*
> - *The data is never, or infrequently, updated.*
>
> *-- Adaptive Server Enterprise 15.5 Performance and Tuning Series: Basics; Chapter 5 Memory Use and Performance; Configuring the data cache to improve performance; Cache replacement strategies and policies*

The first bullet is the most important and is generally quantified as 98% or higher cache hit ratio. The rationale is based on the fact that a strict MRU→LRU chain is not maintained, which means pages used with some frequency may be flushed out of cache as the wash marker may clear their 'recently used' bit and a new page may push them to be a victim before the page is reused. The smaller the cache and more dynamic the cache turnover, the greater the potential is for this to happen. Since the housekeeper doesn't process the cache, with little or no movement of the wash marker, this leaves the primary cache flushing mechanism as the checkpoint process.

The second bullet takes a bit of explanation. It is possible and actually recommended for tables with high volume updates to fixed length fields with no insert activity to be bound to a relaxed cache. For example, the typical sequential key tables (update … set key=key+1) would benefit from being placed in a tiny (1MB) cache with a relaxed cache strategy. So, then, why the warning. Problems start to happen as soon as there is a lot of inserts or once updates start causing page splits, row forwarding – or any activity that causes new pages to be allocated. New pages are always allocated from the LRU end of the chain whether strict or relaxed cache strategy. Since the relaxed cache strategy does not use MRU→LRU reorganization of the cache, it is much more likely that you will hit a cache stall in which the LRU page may have been just updated and is still dirty. This could result in sporadic performance issues that are hard to diagnose.

While a relaxed cache can be partitioned, there are a couple of questions you might want to ask before doing so:

- If the cache turnover is that dynamic (as represented by turnover within the hashtable), should the cache instead have a 'strict' cache replacement policy vs. 'relaxed'?
- Is the contention for the cache spinlock a symptom of the cache warm-up after a server boot and will it disappear or be dramatically lower once the system stabilizes?
- Does the cache contain a table that is experiencing heavy inserts from multiple users that is causing rapid page expansions – and should that table be bound to a different cache?

Generally speaking, a 'relaxed' cache either is not partitioned or only has a few partitions vs. a more normal MRU→LRU 'strict' replacement cache. One consideration for this is the '*global cache partition number*' configuration parameter. Frequently, DBA's will set this to the number of engines and since it applies to all caches, a relaxed cache may be partitioned more than expected. To avoid this, you can individually set the '*local cache partition number*' to a smaller (or higher value) as necessary. An oft forgotten aspect to consider is that if changing the cache replacement policy to 'relaxed', you should also consider changing the '*local cache partition number*' to a lower value if the '*global cache partition number*' is fairly high.

One of the more interesting uses of a 'relaxed' cache is as a transaction log cache. This works despite the fact that pages are constantly being appended to the log due to the fact that pages are static once created and are not normally re-read once flushed from cache. There are exceptions that may cause log pages to be re-read, for example, if the Replication Agent is lagging, it could cause pages to be re-read quite frequently. In addition to the transaction log, another cache that is a good example of when to use a 'relaxed' cache replacement is named caches that contain 'lookup' tables, reference data, as well as caches created to hold text/index columns (indexes) as they frequently are written and rarely read or read only once. Additionally, if a separate cache was created for system tables and if created large enough to hold all the system tables, it might benefit as well. Although for this latter case, you may need to be careful with tables such as sysprocedures/syscomments – which can grow due to procedure re-resolution (sysprocedures) or simply are quite large an cannot completely fit in cache (syscomments).

Typically, you should consider a 'relaxed' cache replacement policy when advised to by sp_sysmon. However, consider the following example:

```
Cache: default data cache
                                per sec      per xact       count  % of total
------------------------- ------------ ------------ ----------- ----------
  Spinlock Contention               n/a          n/a         n/a       1.9 %

  Utilization                       n/a          n/a         n/a      46.6 %

  Cache Searches
    Cache Hits                444812.4       2550.9    14233997      99.9 %
       Found in Wash           30218.0        173.3      966976       6.8 %
    Cache Misses                 361.3          2.1       11562       0.1 %
------------------------- ------------ ------------ -----------
  Total Cache Searches        445173.7       2553.0    14245559

  Pool Turnover
    2  Kb Pool
        LRU Buffer Grab           321.8          1.8       10298      39.0 %
          Grabbed Dirty             0.0          0.0           0       0.0 %
    4  Kb Pool
        LRU Buffer Grab           158.3          0.9        5066      19.2 %
          Grabbed Dirty             0.0          0.0           0       0.0 %
    16 Kb Pool
        LRU Buffer Grab           345.1          2.0       11043      41.8 %
          Grabbed Dirty             0.0          0.0           0       0.0 %
------------------------- ------------ ------------ -----------
  Total Cache Turnover            825.2          4.7       26407

  Buffer Wash Behavior
    Statistics Not Available - No Buffers Entered Wash Section Yet

  Cache Strategy
    Cached (LRU) Buffers       374316.2       2146.6    11978119     100.0 %
    Discarded (MRU) Buffers        41.0          0.2        1313       0.0 %

…
    Tuning Recommendations for Data cache : default data cache
    -----------------------------------
    - Consider using 'relaxed LRU replacement policy'
      for this cache.
```

In this case, it is recommending a ‘relaxed’ cache replacement policy for the default data cache. Normally, this cache would be the one that incurs the most cache turnover. However, if the “hot” tables are bound to separate caches or if the system behavior focuses most of the activity on current data with infrequent historical queries, then changing the default data cache to use a ‘relaxed’ cache policy might be effective. In reality, that is likely a rare occurrence and as a result, not a likely action even though ‘recommended’. One needs to take these recommendations into context. For example, when the cache hit ratio is above a hard coded threshold, the sp_sysmon procedure simply suggests using a ‘relaxed’ cache no matter the contents of the cache.

```
/*
** If the Cache Hit Rate is greater than 95% and
** the replacement is less than 5% and if the
** existing replacement policy is "strict LRU"
** then consider using "relaxed lru replacement"
** policy for this cache.
*/
```

In the above, cache hit rates and buffer replacement are computed via the formula:

```
hit rate % = (buffer search finds * 100.0)/(buffer searches)
replacement % = (buffers washed dirty * 100.0)/(buffer wash throughput)
```

Consider the following example of when it may be the wrong thing to do:

```
-------------------------------------------------------------------------------
  Cache: tempdb_cache
                                per sec      per xact       count  % of total
  -------------------------  ------------  ------------  ----------  ----------
    Spinlock Contention              n/a           n/a         n/a       0.0 %

    Utilization                      n/a           n/a         n/a       1.8 %

    Cache Searches
      Cache Hits                 18082.3           6.3    32783158     100.0 %
         Found in Wash             503.2           0.2      912332       2.8 %
      Cache Misses                 6.9           0.0       12495       0.0 %
  -------------------------  ------------  ------------  ----------
    Total Cache Searches         18089.2           6.3    32795653

    Pool Turnover
      2  Kb Pool
          LRU Buffer Grab           52.9           0.0       95983      22.7 %
            Grabbed Dirty            0.0           0.0           0       0.0 %
      4  Kb Pool
          LRU Buffer Grab          177.8           0.1      322353      76.3 %
            Grabbed Dirty            0.0           0.0           0       0.0 %
      16 Kb Pool
          LRU Buffer Grab            2.2           0.0        3975       0.9 %
            Grabbed Dirty            0.0           0.0           0       0.0 %
  -------------------------  ------------  ------------  ----------
    Total Cache Turnover           232.9           0.1      422311

    Buffer Wash Behavior
      Statistics Not Available - No Buffers Entered Wash Section Yet

    Cache Strategy
      Cached (LRU) Buffers       18090.3           6.3    32797741     100.0 %
      Discarded (MRU) Buffers        0.0           0.0           0       0.0 %

    Large I/O Usage
      Large I/Os Performed         180.0           0.1      326328      99.2 %

      Large I/Os Denied due to
        Pool < Prefetch Size         0.0           0.0           0       0.0 %
        Pages Requested
        Reside in Another
        Buffer Pool                  1.4           0.0        2538       0.8 %
  -------------------------  ------------  ------------  ----------
    Total Large I/O Requests       181.4           0.1      328866

    Large I/O Detail
     4   Kb Pool
        Pages Cached               355.6           0.1      644706       n/a
        Pages Used                   0.0           0.0           0       0.0 %
     16  Kb Pool
```

```
    Pages Cached                17.5           0.0       31800       n/a
    Pages Used                   0.0           0.0           0       0.0 %

Dirty Read Behavior
      Page Requests             128.2           0.0      232366       n/a

Tuning Recommendations for Data cache : tempdb_cache
------------------------------------
- Consider using 'relaxed LRU replacement policy'
  for this cache.
```

First, why was this recommendation made? Answer – because the hit rate percentage was higher than 95% (100% to be exact) and the replacement percentage was 0% (No buffers in wash). Had there been buffer wash activity, the buffer wash behavior section might of looked something like:

```
                            per sec      per xact       count  % of total
-------------------------  ------------  ------------  ----------  ----------

  Buffer Wash Behavior
    Buffers Passed Clean          789.0           0.3     1430487      99.9 %
    Buffers Already in I/O          0.0           0.0           0       0.0 %
    Buffers Washed Dirty            0.0           0.0           1       0.0 %
```

With a buffers washed dirty of 0, even the above case would compute the replacement percentage as 0%. So why is this possibly the wrong thing to do? Think of how tempdb normally works.

- there is a lot of insert activity as new #temp tables are created and populated.
- there are a lot of page splits as rows in #temp tables are updated – frequently from null values to a non-null value
- there is a lot of cache turnover/flushing as new #temp tables are created and others are dropped.

So, just logically, what would happen in a relaxed cache strategy is that the victim page is always going to be active as new #temp tables are created via select/into or insert/selects or rows are updated causing page splits. This is going force the wash marker to constantly move around the buffer ring. Now, ordinarily, this might be construed as a good thing because conceptually, it would the wash marker would go clear around the buffer ring before it got back to the same area just used – and by that time the table may have been dropped. How true this is depends on a number of factors:

- How big the tempdb cache is relative to tempdb usage
- How quickly the #temp tables are dropped after use
- How often the #temp tables are created and read just once vs. read (or updated) multiple times
- The impact of worktables used for query processing
- How many cache partitions there are vs. tempdb cache size

First, we need to realize that the 1.8% for utilization above is not the percentage of tempdb cache utilization, but the percentage of cache searches server wide that occurred in that cache. So, we can't assume from this number that the entire database usage fits in cache. In fact a key indication that it is on the cusp of not fitting in cache is that ~3% of the cache searches found

the page in wash. Not a large number, but still an indication that a good chunk of the cache is being used.

So, here are the problems that a tempdb cache with a relaxed cache strategy could see:

- Because an MRU→LRU chain is not used, there may be higher physical IOs as a result of the wash flushing pages to disk (writes)
- Subsequent reads needing to re-read pages from disk (physical reads) as the cache efficiency is of MRU→LRU chain avoiding writes/reads is lost and pages are flushed.
- In addition to the temp table usage, query processing work tables will also consume temp cache space as well as the tempdb transaction log. The latter should be within the logiosize (4K) pool, so while it is isolated from the data, it does reduce the amount of cache available for the data to use assuming there is a larger IO pool (e.g. 16K) as well. If there is not a larger IO pool, the data pages can use the logiosize (4K) pool.
- Because often the tempdb cache(s) use the default cache partitioning scheme, the wash marker location and movement may be faster than anticipated. Note that the tempdb cache should be partitioned to avoid spinlock contention due to heavy concurrency in tempdb and #temp creation, so this is not avoidable.

A lot depends on how the tempdb IO happens, of course, but unless the tempdb can be largely cached in its entirety, a relaxed cache strategy may not be appropriate. There are times when it would be effective – especially with small tempdb's and multiple tempdbs for OLTP vs. batch. Remember, though, the high occurrence of dynamic table creation/dropping in tempdb is likely going to affect the Object Manager (DES) spinlock as well, so even if a relaxed cache strategy reducing the LRU→MRU relinkage works, the DES contention and cache spinlock contention as new pages added still could be the bottlenecks in prevent larger gains.

Asynchronous Prefetch & Cache Strategy

The final two considerations about general cache tuning are to consider the asynchronous prefetch and the cache strategy being used.

Asynchronous Prefetch

Unfortunately, most DBA's think that asynchronous prefetch (APF) is a positive occurrence. The actuality is that it is a force for good only when contiguous page fetches are unavoidable or are the optimal choice. Too often, APF's are bell ringer for possible table scans or partial table scans that would perform better with appropriate indexing. There also is a tendency to think that APF's will use large I/O's. This can be true if a large I/O pool is defined, but a system with no large buffer pools can still experience APF's. Also, APF's generally only occur on index leaf and data pages due to the way the predictive fetching algorithm works.

APF's are not issued all at once. What happens is actually quite simple. When ASE reads an index page and determines that it will need all of the data or index pages it points to and that those pages are not in cache already, it will issue I/O's for all the pages at once. Alternatively, if

ASE determines that a full or partial table or index scan will be used, it simply issues all the I/Os at once for each extent. If the pages are contiguous and a large I/O buffer pool exists, it will try to use a large I/O. However, not all large I/O's will be used as large I/O's can be denied for a variety of common reasons. The two most common reasons large I/O's are denied is that 1) one of the pages within the page range for the large I/O is already in cache; and 2) buffer pool is smaller than the prefetch size. Consider the following sp_sysmon snapshot:

```
Data Cache Management
---------------------

  Cache Statistics Summary (All Caches)
  -------------------------------------
                                per sec      per xact       count  % of total
                           ------------  ------------  ----------  ----------

    Cache Search Summary
      Total Cache Hits          951290.7        5455.4    30441301       99.5 %
      Total Cache Misses          5042.0          28.9      161344        0.5 %
  -------------------------  ------------  ------------  ----------
    Total Cache Searches        956332.7        5484.3    30602645

…
    Large I/O Usage
      Large I/Os Performed        1715.2           9.8       54885       98.1 %

      Large I/Os Denied due to
        Pool < Prefetch Size        27.7           0.2         885        1.6 %
        Pages Requested
        Reside in Another
        Buffer Pool                  4.9           0.0         157        0.3 %
  -------------------------  ------------  ------------  ----------
    Total Large I/O Requests      1747.7          10.0       55927

    Large I/O Effectiveness
      Pages by Lrg I/O Cached     9037.5          51.8      289200        n/a
      Pages by Lrg I/O Used       9011.5          51.7      288368       99.7 %

    Asynchronous Prefetch Activity
      APFs Issued                  234.9           1.3        7518        2.1 %
      APFs Denied Due To
        APF I/O Overloads            0.0           0.0           0        0.0 %
        APF Limit Overloads          0.0           0.0           0        0.0 %
        APF Reused Overloads         0.0           0.0           0        0.0 %
      APF Buffers Found in Cache
        With Spinlock Held          11.8           0.1         379        0.1 %
        W/o Spinlock Held        10766.2          61.7      344519       97.8 %
  -------------------------  ------------  ------------  ----------
    Total APFs Requested         11013.0          63.2      352416

    Other Asynchronous Prefetch Statistics
      APFs Used                    216.8           1.2        6939        n/a
      APF Waits for I/O            104.6           0.6        3348        n/a
      APF Discards                   0.0           0.0           0        n/a

    Dirty Read Behavior
      Page Requests                 14.2           0.1         453        n/a
```

Note that the large I/O's were a factor of 9x higher than the APF's issued. There can be a variety of reasons for this. In addition to DBA utilities such as create database, dbcc, etc., large I/O's also can be used by bulk inserts as well as select/into. As a result, as long as there is a 16K pool in the cache where tempdb is located, common select/into #temp will use large I/O's to populate the table.

In the above, no APF's were denied due to limits on APF's. Normally, this might be taken as a good thing. However, the presence of APF's is a cause for further investigation. Because an APF can make use of a large I/O pool, the optimizer often times will make a determination whether an index access method would be cheaper than a table scan. As a result, the optimizer might decide to use a table scan as the large I/O coupled with APF's may have a predicted faster response time. This can problematic and misleading:

- The table scan increases the I/O load on the system and may actually have a slower response time due to I/O issues or capacity.
- The table scan would use shared locks on all the pages – possibly resulting in higher contention with other users
- As mentioned, the positive connotation of APF's may be masking the fact an unnecessary table scan is occurring due to a missing index or poor index selectivity.

In addition to the above considerations, there is another aspect – cache turnover. We all know that ASE tends to choose a table scan when it estimates that more than 40% of the rows will be accessed. However, that rule is not the one the governs APF's with large I/O's – which is based purely on the normal costing of physical I/O's + logical I/O's + CPU time as reported as the 'total cost' of a query.

Some may misinterpret this discussion as anti-APF – it is not. APF's can improve performance – especially when used in conjunction with large I/O pools. However, in the past DBA's have had the tendency to associate that APF's were *always* a good thing, while in reality, closer inspection is needed to determine if really beneficial, if over eagerly invoked or if just masking a problem. If overeagerly invoked, you can control it individually at the table level, session level or cache level.

Optimizer induce APF's are most common on smaller tables. We all are familiar with the old (and not necessary accurate) ASE adage that ASE will always use a tablescan if the table has fewer than 10 pages. Part of this adage stems from APF's. For example, assume we are after two different rows on two different pages in a small table of 8 pages. If the table is not already in cache we would have to do minimally 3 physical IO's – one for the index (it would be small enough to fit on a single page probably), and then 1 for each of the rows. Total cost is 3 physical IO's and 3 logical I/O's. If an APF prefetch is allowed and a large IO pool exists, the server could read the entire table in a single large I/O but then would need 8 logical reads to find the qualifying rows. The difference (ignoring CPU) in cost is:

```
Index: 3*25+3*2 = 75+6 = 81
APF tablescan: 1*25 + 8*2 = 25 + 16 = 41
```

Consequently, the tablescan appears a lot cheaper. The problem could be those 8 logical reads – only two of the pages are necessary. The extra locking on the other 6 pages could increase system contention.

The point is that APF's and usage of large I/O's are often thought of as purely *physical* I/O implementations. This gets confusing as the optimizer determines whether to use APF's prior to checking to see if the pages are in cache. As a result, some APF attempts are satisfied from cache. This can be seen quite clearly from sp_sysmon:

```
  Asynchronous Prefetch Activity
    APFs Issued                   403.5           0.1      731455       0.8 %
    APFs Denied Due To
      APF I/O Overloads             0.0           0.0           0       0.0 %
      APF Limit Overloads           0.0           0.0           0       0.0 %
      APF Reused Overloads          9.3           0.0       16882       0.0 %
    APF Buffers Found in Cache
      With Spinlock Held           18.2           0.0       32999       0.0 %
      W/o Spinlock Held         51987.6          18.2    94253504      99.2 %
  ------------------------  ------------  ------------  ----------
  Total APFs Requested          52418.6          18.4    95034840

  Other Asynchronous Prefetch Statistics
    APFs Used                     389.2           0.1      705670       n/a
```

```
APF Waits for I/O             154.1           0.1      279368       n/a
APF Discards                    0.0           0.0           0       n/a
```

In the above case, almost all of the APF prefetch attempts were satisfied from cache. Of those that were issued, ~33% had to wait for IO – and probably was visible as WaitEventID 124.

Regardless, a table that is experiencing APF prefetches in a normal (strict) cache will have a high impact on the LRU-MRU page relinking. Additionally, often APF prefetches are a sign that the query criteria was fairly broad or not selective enough for index coverage and the result is that ASE believes it to be faster to do large I/O's and even read pages it doesn't need vs. individual I/O's for just the necessary pages. Consequently, a table that is experiencing a lot of APFReads (along with PhysicalReads) as measured by monOpenObjectActivity may be experiencing higher cache turnover than it should be. One possible cause is that the cache strategy is using an MRU strategy (next section) and as a result, is flushed from cache quicker – but the more likely cause is that the extra unnecessary pages is forcing other pages out of cache. However, it also is a possible sign that the cache configurations may not be set up correctly. If the system is memory limited, there may not be anything you can do necessarily about the cache turnover. However, by reducing the cache's 'asynchronous prefetch percent' from the default of 10% to something considerably smaller, you might be able to prevent some of the tablescans due to perceived lower total query cost – especially if the system is experiencing I/O performance issues or contention. For DOL tables, changing the 'opt concurrency threshold' also helps as it can reduce the tendency of ASE to use tablescans on smaller tables. One way to prevent it entirely would be to disable prefetching at the session or for the table by using sp_cachestrategy or sp_chgattribute:

```
-- disable prefetching for the session
set prefetch ["on" | "off"]

-- Permanently disable prefetch for a table:
exec sp_cachestrategy <dbname>, <tablename> [,<indexname> | "text only" | "table only"],
        "prefetch", ["on" | "off"]
go

-- Prevent tablescans when concurrency is above a threshold (DOL):
exec sp_chgattribute <tablename>, "concurrency_opt_threshold", <num_pages>
```

The question is which do you use and when. If you want to prevent optimizer triggered tablescans or indexscans on a particular table entirely, you would use sp_cachestrategy. If used on a small table (or example 8 page table), it would always enforce index access – which might help reduce contention on volatile small tables. The problem with this is that it would also prevent tablescans/indexscans where they would be useful (range queries) and would hinder the performance of tablescans or indexscans where the optimizer didn't have a choice. Consequently, sp_cachestrategy is a good fit for small, volatile APL tables – but should be used with care on tables that could experience range scans.

On the other hand, sp_chgattribute may be a better choice for DOL tables. If the number of pages in the table is less than the pages specified in sp_chgattribute, ASE will use the indexes vs. an APF tablescan. This prevents tablescans on smaller tables where concurrency on pages read but not necessary for the query is causing contention. The default value of 15 is interesting in that it prevents APF induced tablescans on tables less than 15 pages – which is one of the reasons why the old adage about 10 pages always using a tablescan is inaccurate. Note, that this applies for DOL tables only. This might appear to be an odd restriction, but the focus here is on concurrency and contention caused by APF tablescans. In such cases (high concurrency/high contention), the first consideration should be to change from APL to DOL

locking. Some also might see an issue with the fact that the max for concurrency_opt_threshold is 32767, although -1 can be specified to prevent tablescans on tables with more than 32767 pages. However, the reality is by the time a table gets more than 1000 pages in it, index access will generally be much cheaper unless the number of rows hits the 40% threshold that will trigger a tablescan regardless. As a result, it is probably more advisable to start off with using sp_chgattribute "concurrency_opt_threshold" and then supplement it by using sp_cachestrategy on specific indexes on smaller tables. You should take care to ensure the sp_cachestrategy is not used to disable prefetching on indexes that normally could be used in a range scan – such as indexes based on date values or names.

Cache Strategy

The term "MRU strategy" is a bit of a misnomer – the more accurate "fetch and discard" better describes what is happening. Many people think that with an MRU strategy, that ASE reuses the page at the MRU end of the chain vs. caching the data – and therefore only one page of the table or index will be in cache at a time. The result (of course) is that a large tablescan won't flush other data out of cache. What happens is actually quite different.

When an MRU strategy is used, rather than placing the page at the front of the MRU in the MRU→LRU chain, the page is placed at the wash marker. Each subsequent page is also placed at the wash marker – forcing the preceding pages to enter the wash area. There are several aspects to this:

- Depending on the wash size and cache volatility, substantial amounts of a table scanned using an MRU strategy could end up in cache. By default this could be up to 60MB in larger caches/buffer pools.
- Data already in the wash area would be in danger of getting flushed out.
- Data not in the wash area would be protected from being flushed out

In other words, rather than thinking of MRU as reusing the same page over and over, what it really is doing is reusing the wash area pages in a cyclical fashion.

Generally, most DBA's associate the MRU strategy with tablescans. While this is true, it is not the only time when it occurs. Consider the following:

- MRU will be chosen when optimizer estimates that 50% of the buffers in a buffer pool will need to be replaced as a result of using a normal LRU strategy
- MRU will be chosen on an inner table of a join using a unique, ascending key. The rationale is that once the inner table page has been read once, it will not be needed again for that query.
- MRU will be chosen for the outer table of a join in which the outer table must be scanned. Similar to above, the rationale is that each page of the outer table would be used one time.
- MRU does not apply to index root or intermediate nodes (only leaf nodes and data pages)
- MRU does not apply to pages already in cache

- MRU does not apply to dirty pages (which would already be in cache)

Generally, the ASE optimizer is very good at determining the MRU vs. LRU strategy. However, if you find that you have a table that almost always is the inner table of a join and the join is on an ascending key – or if the outer table and scanned a lot, the MRU strategy may be contributing to a higher than necessary physical read load. It is tough to isolate this problem, however, as MRU vs. LRU strategy is a buffer pool related concept whereas monOpenObjectActivity and other MDA tables that provide object level stats don't track this information. However, if you suspect this is the case, one possible method would be to use monCachedObject and monitor how much the object uses in the cache it is bound to. If the amount cached rarely exceeds the wash size for the data pages, you can try disabling the MRU strategy with the sp_cachestrategy procedure as well.

```
-- Permanently disable MRU/fetch-and-discard strategy for a table:
exec sp_cachestrategy <dbname>, <tablename> [,<indexname> | "table only"],
        "mru", ["on" | "off"]
```

This should be done very carefully, followed by careful monitoring via monCachedObject of other tables in that cache to make sure that the disable of the MRU strategy on one table doesn't cause others to be bumped out of cache too frequently. There will be some impact of course, you just don't want it unnecessarily hindering performance.

All of the above cache tuning strategies are used in ASE when setting up named caches. The following sections discuss the different caches, which ones are recommended – and the recommended cache tuning strategies as a starting point for each.

Default Data Cache

The default data cache is generally the cache from which the memory is carved out for all the other caches. However, it likely is still the largest cache in the system and contains the bulk of the data. If the default data cache is not the largest in the system, it suggests a possible problem. There are exceptions to this rule – for example, creating separate named caches for different databases that have equivalent application workloads and similar application workload profiles such as peak processing times, etc. However, once you begin to move tables or databases out of the default data cache, you now need to pay considerable attention to cache usage, turnover and other statistics to ensure that the caches are effectively utilized.

Keep in mind that ASE workload management currently seems to be best described as *"isolate the good and the bad; ignore the rest"*. In other words, from a cache standpoint, you want to isolate the hot data tables to ensure they remain in cache, while also isolating other tables (or indexes/text chains) to prevent them from consuming too much cache. What remains after that usually does not have enough of an impact that tuning it will have substantial payback – consequently, largely ignorable as the effort vs. payback is not conducive.

As far as tuning the default data cache, the following recommendations are made *only* as a *starting* position.

Cache Configuration/Tuning	Comments/Setting
Size	As large as possible after all other memory requirements plus "cushion"
Buffer pools	70% in pagesize; 30% in 8x pagesize; 50-100MB in 2x pagesize. For example, if a 2K server and a 20GB default data cache, you would start with 6GB in 16K pool, 100MB in 4K pool and the rest in 2K pool.
Partitions	Max engines online/2 rounded to nearest power of 2. For example, if 20

	engines are maximum, 20/2=10 rounded to power of 2 = 8.
Relaxed or Fixed cache strategy	Default (fixed)
HK Ignore Cache	NO
Prefetch percentage	10% or less depending on APF induced occurrences of tablescans.
MRU strategy	As needed per object, but otherwise no.

Procedure Cache

The next major pool of memory to be configured at the very beginning is the procedure cache. In reality, this is a bit of a misnomer as the procedure cache in ASE contains:

- cached query plans for the statement cache
- cached query plans for stored procedures
- subquery result cache
- index histogram statistics
- auxiliary sort buffers (the sort itself takes place in the data cache where the data resides (most often tempdb's cache) – this is memory used to track the sort
- scrollable cursor position information
- in-memory sorts

And so on. Literally, it is everything but data cache. Most of the above list has always been in procedure cache – just DBA's didn't account for it as it wasn't as well known.

In ASE 15.x, it became more crucial to size procedure cache due to the increase in cache used for index histograms due to the use of 'update index statistics' and the change in the default 'histogram tuning factor'. The old methods of sizing proc cache usually used some form of estimating the number of concurrent stored procedures multiplied by an average procedure size. Unfortunately, this is only a small part of the proc cache. A better *rough* estimate might be:

```
est_proc_cache = sp_spaceused(systabstats)+(concurrent_users*10*16KB)
                 +(6MB * sort_buffers * concurrent index maintenance)
```

The best way is to start with an estimate and then monitor with the MDA tables monCachedProcedures, monProcedureCache, monStatementCache and monCachedStatements.

Stored Procedure Cache Usage

In most systems, there is a small subset of "hot" stored procedures that are used the most frequently – no matter how many procedures exist in the system. This is followed by a wider set of procedures that are used often, but are not executed quite as frequently. And of course, there is even a wider set of procedures that are rarely used at all. Ideally from a procedure caching point, the following should be the goal:

"Hot" Procedures – these procedures should have enough concurrent plans in cache that the only ones that have reads from disk are those with execute with recompile. The number of plans in cache should remain fairly stable.

"Frequent" Procedures – these procedures should have enough concurrent plans in cache that very few reads from disk are required – a max of 1-2 procedure reads per second – fewer if manageable.

"Rare" Procedures – since these procedures are rarely used the number of reads from disk are not relevant.

Unfortunately, procedure cache is not controllable. If a "rare" procedure is needed to be loaded into proc cache, you can't specify for it to flush another recently used "rare" procedure – the system may simply pick one of the frequently executed ones – or worse yet, one of the hot procedures. You also can't control when it needs to load statistics that it doesn't flush a hot proc – or vice versa – that when it loads rarely executed proc, that it doesn't flush index statistics that are used frequently. The only good thing is that procedure cache is managed on a MRU→LRU chain for the proc headers – but you could still have issues much as with a single data cache in this respect. Consequently, you have to manage the proc cache size so that the above guidance is maintained.

The easiest way to do this is to monitor monProcedureCache and monCachedProcedures. Consider the following table definitions for these MDA tables:

```
create existing table monProcedureCache (
        Requests                        int,
        Loads                           int,
        Writes                          int,
        Stalls                          int,
        InstanceID                      tinyint,
)
materialized at "$monProcedureCache"
go

create existing table monCachedProcedures (
        ObjectID                        int,
        InstanceID                      tinyint,
        OwnerUID                        int,
        DBID                            int,
        PlanID                          int,
        MemUsageKB                      int,
        CompileDate                     datetime,
        ObjectName                      varchar(30) NULL,
        ObjectType                      varchar(32) NULL,
        OwnerName                       varchar(30) NULL,
        DBName                          varchar(30) NULL,
        RequestCnt                      int NULL,
        TempdbRemapCnt                  int NULL,
        AvgTempdbRemapTime              int NULL,
)
materialized at "$monCachedProcedures"
go
```

While many may think that monProcedureCache.Stalls is critical, the most important item to watch is the monProcedureCache.Loads. If you are loading several procedure plans per second, it is likely that you have very high turnover and that proc cache is not quite large enough. This can be confirmed by running a query similar to the following (assumes MDA data collected to a repository):

```
-- query to retrieve the number of plans per interval
select SampleTime, DBID, OwnerName, ObjectName, NumPlans=count(PlanID),
        OldestPlan=min(CompileDate), NewestPlan=max(CompileDate)
  from monCachedProcedures
  group by SampleTime, DBID, OwnerName, ObjectName
  order by DBID, OwnerName, ObjectName, SampleTime

-- query to retrieve the total number of plans used.  This really
-- should be done in such a way to find new plans per interval (where
-- the CompileDate is greater than the previous SampleTime).  That query
-- would be a little difficult to document here
select DBID, OwnerName, ObjectName, NumPlans=count(PlanID),
        OldestPlan=min(CompileDate), NewestPlan=max(CompileDate)
  from monCachedProcedures
```

```
group by DBID, OwnerName, ObjectName
order by DBID, OwnerName, ObjectName
```

Care must be taken when running this query. As with any query against monitoring tables that track individual object level statistics, too frequent polling of the table could impact performance. For example, running the above query once every few minutes is not a problem. Attempting to run it every second or every 5 seconds will likely lead to issues. It can be useful for graphing over time such as the following graph:

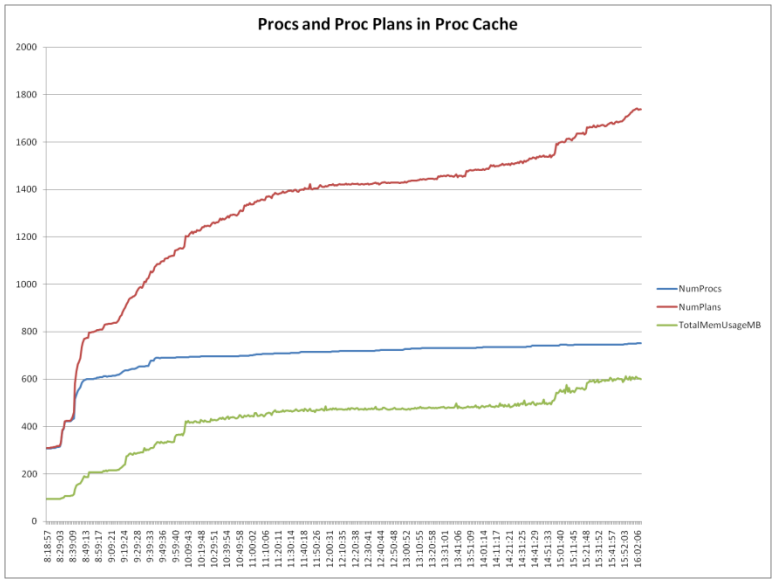


Figure 18 – A Graph Depicting Procedure Cache Usage by Procedure Plans

What you are looking for is any wide variance in the number of plans in memory. The above is fairly consistent with respect to the number of stored procedures in cache and on average about 2 cached plans per procedure (although the actual range was from 1 to 30). The bump at the end was caused by contention on one table resulting in a higher degree of concurrency for one procedure and an increase in the number of plans for that procedure – and memory used by stored procedures but without an increase in the actual number of procedures in cache. Instead of steady as above, if the number of "hot" or "frequent" procedure plans fluctuate much and the number of monProcedureCache.Loads > 1-2 per second per engine, you likely need to increase procedure cache or find out what is causing all the loads. By "Loads", we are referring to non-recompile induce loads – which unfortunately is only tracked by sp_sysmon. Consider the following snippet:

```
Procedure Cache Management      per sec            per xact          count     % of total
---------------------------     ------------     ------------     ----------     ----------
Procedure Requests                    4847.9              1.4         290875            n/a
Procedure Reads from Disk              510.9              0.1          30652         10.5 %
Procedure Writes to Disk                 0.1              0.0              3          0.0 %
Procedure Removals                    1019.7              0.3          61182            n/a
Procedure Recompilations               508.0              0.1          30477            n/a

Recompilations Requests:
    Execution Phase                    507.9              0.1          30471        100.0 %
    Compilation Phase                    0.1              0.0              6          0.0 %
    Execute Cursor Execution             0.0              0.0              0          0.0 %
    Redefinition Phase                   0.0              0.0              0          0.0 %

Recompilation Reasons:
    Table Missing                      503.9              0.1          30234            n/a
    Temporary Table Missing            503.9              0.1          30234            n/a
    Schema Change                        4.1              0.0            243            n/a
    Index Change                         0.0              0.0              0            n/a
```

```
Isolation Level Change          0.0             0.0             0           n/a
Permissions Change              0.0             0.0             0           n/a
Cursor Permissions Change       0.0             0.0             0           n/a
```

In the above example, it appears as there is a lot of procedure reads from disk – and there are. However, most of them are due to procedure recompilations due to temp tables 'missing' (which really means the #temp table existed outside the proc, the proc was compiled with it and then the #temp table dropped…a common technique that is often used for subprocs and temp tables as well as other possibilities). So the real procedure reads from disk is only ~3 per second which is likely not too bad depending on the number of engines.

Note that a procedure may also be flushed out of cache due to object descriptor (DES) reuse. If you see a sudden change in number of plans from some number to zero and then back up but the loads seem to imply nothing else was loaded, this might be the problem. If so, using dbcc tune(des_bind) also works on stored procedures. Note that it locks the procedure's object descriptor in memory – not the number of plans.

Sort Buffers

A common issue in ASE 15.0+ is the amount of procedure cache used for sort operators. The reason is that prior to ASE 15, the optimizer didn't have too many in-memory sort or hash operators and now it does. As a consequence, the amount of procedure cache that needs to be available for sort buffers has increased significantly. Compounding this problem in ASE 15.0 is the lack of statistics or the lack of indices on key join columns or predicates that results in one of the newer sort/hash operators being used to reduce the logical and physical IOs.

Even in 12.5, however, more procedure cache was used for sorts than likely was anticipated. One of the common configuration settings used in 11.x and 12.x ASE engines was the 'number of sort buffers'. In 11.x and 12.x, this had a very positive impact and consequently often was adjusted to a fairly large number such as 5000, 10000 or even 20000. Unwittingly, one of the side effects of this was that proc cache used during stored procedure calls was extremely high due to another common situation – creating indices on #temp tables. The impact can be seen in the following chart:

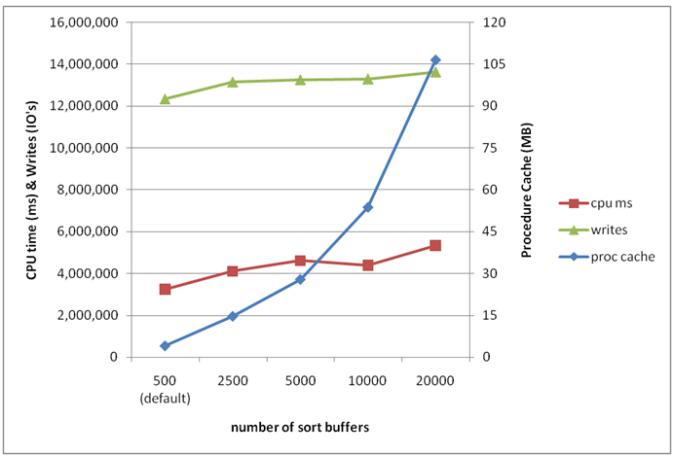
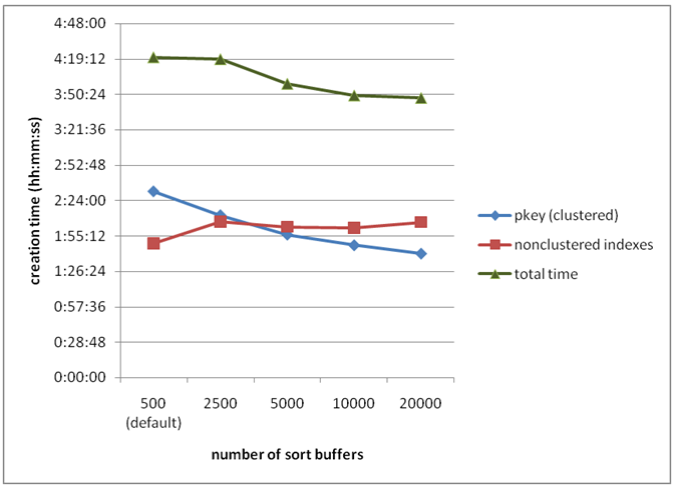


Figure 19 – Procedure Cache Usage vs. Number of Sort Buffers for Index Creation on 250M Rows

The above may be a little extreme for #temp tables inside stored procedures as few #temp tables approach 250 million rows – although some nightly batch processes on larger systems often hit in the high 10's of millions of rows. By leaving the number of sort buffers set at a high value, the result could be large procedure cache consumption – especially if the procedure is being executed concurrently.

The really interesting fact about this is that in ASE 15, the number of sort buffers doesn't seem to help non-clustered indices and only really appears to aid on clustered indexes. Consider the following chart for the same create indexes on the same table as the above chart:



The reason this is interesting is most of the indexes created on #temp tables in stored procedures are non-clustered indexes. As a result, the best method is to simply leave the 'number of sort buffers' at a low setting and only when desired dynamically increase it to a larger value – resetting it to the lower value when finished.

Statement Cache

The statement cache is a critical component of improving overall ASE scalability. However, it really only functions if both the statement cache is created *and* literal autoparameterization is enabled. That aspect is key.

Statement Cache vs. Fully Prepared Statements

To better understand why the statement cache is important, it helps to understand the role of application development API's. In most common API's, there are three basic statement execution interfaces that can be invoked by developers:

- Static SQL language statement interface – unfortunately, this is probably the most common interface. SQL statements are constructed as strings and submitted to the DBMS as language commands.
- Stored Procedure RPC interface – this interface is fairly common for invoking stored procedures using the RPC method in which the client application sends the proc to the DBMS using the RPC interface. In the RPC interface, the client is responsible for datatype translation

- Fully Prepared Statement interface – this interface is best for OLTP applications. In this interface, a parameterized statement is created in both the client and server. The statement is executed by simply invoking the statement number with the required parameters.

Unfortunately, while the Fully Prepare Statement interface is best for OLTP applications, very few OLTP applications actually employ it due to lack of programmer's knowledge of the full import of the interface. The important aspect is that the prepared statement on the DBMS server is implemented in ASE as "Light Weight Procedure" (LWP) – which like any stored procedure is precompiled and pre-optimized. As a result, execution of a fully prepared statement is extremely fast as the usual parse, compile, optimization steps are avoided. For simple single table DML statements and queries, the parse, compile and optimization steps often take up the bulk of the time – and certainly the bulk of the CPU used by the query. This is also true for quick "point" queries that may involve multiple joins but only return a few rows. It is not uncommon for an OLTP application that was using language commands to experience a 5-10x increase in throughput once fully prepared statements were used.

Because of the lack of use of fully prepared statements, modern DBMS's such as Sybase ASE and Oracle both implement a statement cache. The statement cache reproduces the same benefits as fully prepared statements as it takes language commands from client applications, replaces literals with parameters, creates a statement hash key, compiles and optimizes the statement, and creates a Light Weight Proc for re-use. Subsequent language commands are parsed to a hash key and if the hash key matches, the stored pre-optimized query plan is executed. Effectively, the statement cache overcomes the shortcomings of programmers who fail to use fully prepared statements.

Statement Cache Issues

However, the statement cache is not without its problems. One of the most common issues is simply due to sloppy coding – specifically sloppy type casting in literals. Consider the following two statements:

```
select name from sysobjects where id = 1
select name from sysobjects where id = 1.0
```

Their parameterized versions are:

```
select name from sysobjects where id = @@@V0_INT
select name from sysobjects where id = @@@V0_NUMERIC_2_1
```

The result is that due to the different datatypes, the second statement does not match the first. Rather than using the pre-optimized plan for the first statement, the second statement creates a query plan that is added to the statement cache. This isn't so bad when the literals are scalar predicates – the problem is when the query contains an "IN" clause as the possible number of permutations of plans can get unwieldy.

The second problem is nearly as common but hard to identify as few are familiar with monitoring the statement cache or how to control the statement cache to fix the problem. Consider the following queries:

```
Select * from table where date_col between 'Jan 1 2006' and 'Jan 31 2006'
Select * from table where date_col between 'Jan 1 2006' and 'Dec 31 2006'
```

Assuming an index on date_col, if the first query is executed first, the cached plan would likely use the index. When the second query arrived, the query hash key would be matched at it

would use the cached plan (using the index). The problem is that a table scan might be much more efficient for the second query. Worse yet, if the second query was executed first, the cached plan would likely include a table scan. When the first query would be run, it too would use a table scan based on the cached statement. This is very similar to the age-old problem with stored procedure parameters, with a notable difference – there is no "exec with recompile" equivalence. As a result, it is important that systems using the statement cache to exploit bypassing the optimizer to monitor the statement cache for bad query plans.

Statement Cache & Scalability

One of the biggest problems with the statement cache is the fact that it is either on or off – no in-between. For example, from the description above with respect to statement cache vs. fully prepared statements, it should be obvious that the statement cache should only be used for extremely frequently executed statements – and primarily insert, update, delete or very short execution selects as noted earlier. In other words, it should only be used *in place* of fully prepared statements – where fully prepared statements should have been used. While it can be turned on or off at a session level to enable statement caching within scripts or modules of an application, this assumes that application developers will add the session level commands to their applications – which while doable – is not as effective as having them use fully prepared statements instead.

As a result, the statement cache should be fairly small (comparatively) and have as little volatility as possible. The reason why is that an extremely large statement cache or one with a lot of volatility can significantly impede scalability. As with any cache, the statement is hashed using a hash key – in this case MD5 due to possible size of the SQL text. Then the corresponding hash bucket is retrieved and then serially scanned for a matching statement – very similar to the discussion on lock chains. With a limited number of pre-defined hash buckets, the larger the statement cache, the longer this serial scan will take – much as with a small number of hash buckets in the lock hash table. The comes a point where the serial scan for a statement is longer than the optimization would have taken.

Much as with a lock chain, when a new statement is cached (and typically one removed to make room for it), the memory has to be protected from concurrent access with a spinlock. The result is that with high statement cache volatility, the statements that need to be cached or cache searches all are impeded by the spinlock contention. This is one reason why 'literal parameterization' is so crucial. Without it, each OLTP update and delete will likely specify different primary keys – adding new statements to the cache – with little or no reutilization. Even the fast polling style queries often have different literals – also adding to the cache volatility.

As a result, the following recommendations are made with respect to the statement cache:

- Start with a small statement cache – i.e. 10,240 pages.
- Exceeding 100MB is likely far too much. 20-40MB should be the limits.
- Make sure 'enable literal autoparameterization' is on
- Use a login trigger or session commands on the SQLINITSTRING connection property to turn on or off the statement cache based on the application profile.

By monitoring the statement cache, you can determine when the statement cache truly undersized or when the volatility is due to the wrong users/applications having it enabled. Unless the statement is executed 100's of times per minute and is sustained at that rate – it isn't an effective use of statement cache to cache the SQL. As mentioned, you want to see very high cache hit ratios and number of executions.

Monitoring Statement Cache

There are two levels for monitoring statement cache. The first is the high level that DBA's may be familiar with from the typical sp_sysmon report. Consider the following fragment:

```
Procedure Cache Management        per sec      per xact       count  % of total
--------------------------  ------------  ------------  ----------  ----------
…
  SQL Statement Cache:
    Statements Cached                 0.0           0.0           0       n/a
    Statements Found in Cache         0.0           0.0           0       n/a
    Statements Not Found              0.0           0.0           0       n/a
    Statements Dropped                0.0           0.0           0       n/a
    Statements Restored               0.0           0.0           0       n/a
    Statements Not Cached          3445.4           1.2     6246592       n/a
```

However, a much better method to get high level information is from the MDA tables. Consider:

```
create existing table monStatementCache (
        InstanceID                      tinyint,
        TotalSizeKB                     int,
        UsedSizeKB                      int,
        NumStatements                   int,
        NumSearches                     int,
        HitCount                        int,
        NumInserts                      int,
        NumRemovals                     int,
        NumRecompilesSchemaChanges      int,
        NumRecompilesPlanFlushes        int,
)
materialized
at "$monStatementCache"
go
grant select on monStatementCache to mon_role
go
```

From this we can see some interesting statistics that might help with configuration:

TotalSizeKB vs. UsedSizeKB – provides a good indication if the statement cache is too big or too small.

UsedSizeKB/NumStatements – provides a good estimate for the amount of statement cache used per statement when adjusting the statement cache size. This is especially useful if the tables are much wider or statements more complex than the rule of thumb of each statement being 1-2KB in size.

HitCount/NumSearches & *HitCount/NumStatements* – both of these metrics provide an indication of how effective the proc cache is. The first from an overall perspective as a 92% hit count ratio is obviously much better than 10%. The second gives a good indication of overall average reuse for each statement – a 10:1 ratio obviously is much better than 2:1. Note that the sampling interval plays a huge role in this last metric – during a 1 second sample, not that many statements may be executed as compared to a 10 minute sample – and could distort the ratio to be viewed as excessively low.

NumInserts & NumRemovals – both of these are good indications of the turnover within the statement cache. If the UsedSizeKB is close to the TotalSizeKB and

the turnover is fairly high, it could be an indication that an application is not as effectively using statement caching as it could – either due to the wide range of queries – or due to impact of sloppy type casting, etc. This requires more detailed monitoring to really determine.

But all of these just give a sense of the overall health of the statement cache. Much like any data cache or procedure cache, it can have metrics that appear healthy (i.e. 95% hit rates) but those numbers can be skewed heavily by bad activity – or meaningless. For example, a 100% hit rate on a statement cache that has cached all table scans is not a good thing. Because of that, it only makes sense to use monStatementCache monitoring for determining statement cache sizing and not as a good indication of the overall effectiveness of statement caching itself. Instead, a closer monitoring of the individual cached statements is required.

```
create existing table monCachedStatement (
        InstanceID                    tinyint,
        SSQLID                        int,
        Hashkey                       int,
        UserID                        int,
        SUserID                       int,
        DBID                          smallint,
        UseCount                      int,
        StatementSize                 int,
        MinPlanSizeKB                 int,
        MaxPlanSizeKB                 int,
        CurrentUsageCount             int,
        MaxUsageCount                 int,
        NumRecompilesSchemaChanges    int,
        NumRecompilesPlanFlushes      int,
        HasAutoParams                 tinyint,
        ParallelDegree                tinyint,
        QuotedIdentifier              tinyint,
        TransactionIsolationLevel     tinyint,
        TransactionMode               tinyint,
        SAAuthorization               tinyint,
        SystemCatalogUpdate           tinyint,
        MetricsCount                  int,
        MinPIO                        int,
        MaxPIO                        int,
        AvgPIO                        int,
        MinLIO                        int,
        MaxLIO                        int,
        AvgLIO                        int,
        MinCpuTime                    int,
        MaxCpuTime                    int,
        AvgCpuTime                    int,
        MinElapsedTime                int,
        MaxElapsedTime                int,
        AvgElapsedTime                int,
        DBName                        varchar(30) NULL,
        CachedDate                    datetime NULL,
        LastUsedDate                  datetime NULL,
        LastRecompiledDate            datetime NULL,
)
materialized
at "$monCachedStatement"
go
```

There are several levels of monitoring in this respect. The first level is just sensing when a particular cached statement has likely cached a bad query plan. For this, there are several bell-ringer “warning bell” metrics to consider:

- AvgLIO & MaxLIO
- AvgPIO & MaxPIO
- AvgCpuTime & MaxCpuTime
- AvgElapsedTime & MaxElapsedTime

While IO related metrics are useful for spotting table scans, the best of these is the time related metrics. Part of the reason for this is that in-memory sorts may often reduce the number of

IO's making the plan look good from an IO perspective, but actually increase the amount of CPU usage beyond what is necessary. The order of concern is best determined by deriving the total CPU or total elapsed time. Consider the following query fragment:

```
-- Query for ASE 15.5+, for ASE/EE (SMP..non-CE) 15.0.x,
-- remove the InstanceID checks (first predicate)
select …
        TotCpuTime = AvgCpuTime * MetricsCount,
        TotElapsedTime = AvgElapsedTime * MetricsCount,
        …,
        query_text=show_cached_text(SSQLID)
  from master..monCachedStatement
  where (@@instanceid is null or InstanceID=@@instanceid)   -- remove if ASE/EE <15.5
    and (AvgCpuTime > 500 or AvgElapsedTime > 3000)
  order by AvgCpuTime * MetricsCount desc
-- order by AvgElapsedTime * MetricsCount desc
```

Obviously, this can be adjusted as necessary to filter on small or larger averages or changed to focus on longer running queries vs. those consuming CPU. The totals help you not only identify the bigger problem queries but also help normalize the average vs. max problem. If the max is fairly close to the total, then you know that a single or a few large exceptions caused the average to be distorted and if you subtract the max from the total, you can compute a better average for normal executions. However, if the max is quite far from the total and much closer to the average, then you know that it is the normal executions that are driving the overall time or I/O related metrics.

Statement Cache Controls

In the above query fragment, you might have noticed the use of the show_cached_text() function. This function returns the query text in its parameterized form – or example:

```
select * from authors where au_id in (select au_id from titleauthor where au_ord > @@@V0_INT) and
au_id not in (select au_id from titleauthor where au_ord=@@@V1_INT)(@@@V0_INT INT, @@@V1_INT INT)
```

Sometimes, merely seeing the query is enough to filter out queries that would run for a long time simply due to the number of rows that they typically return or due to query complexity. However, to verify that the cached statement is optimal, you need to look at the query plan using show_plan() function:

```
-- show_plan(-1,SSQLID,-1,-1)
1> select show_plan(-1, 884859206, -1, -1)
2> go

QUERY PLAN FOR STATEMENT 1 (at line 0).


    STEP 1
        The type of query is DECLARE.

Total estimated I/O cost for statement 1 (at line 0): 0.


QUERY PLAN FOR STATEMENT 2 (at line 1).


    STEP 1
        The type of query is SELECT.

         7 operator(s) under root

       |ROOT:EMIT Operator (VA = 7)
       |
       |   |MERGE JOIN Operator (Join Type: Inner Join) (VA = 6)
       |   | Using Worktable1 for internal storage.
       |   |  Key Count: 1
       |   |  Key Ordering: ASC
       |   |
       |   |   |SQFILTER Operator (VA = 4) has 2 children.
       |   |   |
       |   |   |   |GROUP SORTED Operator (VA = 1)
       |   |   |   |Distinct
```

```
|   |   |   |
|   |   |   |   |SCAN Operator (VA = 0)
|   |   |   |   |  FROM TABLE
|   |   |   |   |  titleauthor
|   |   |   |   |  Table Scan.
|   |   |   |   |  Forward Scan.
|   |   |   |   |  Positioning at start of table.
|   |   |   |   |  Using I/O Size 4 Kbytes for data pages.
|   |   |   |   |  With LRU Buffer Replacement Strategy for data pages.
|   |   |   |
|   |   |   |  Run subquery 1 (at nesting level 1).
|   |   |   |
|   |   |   |  QUERY PLAN FOR SUBQUERY 1 (at nesting level 1 and at line 3).
|   |   |   |
|   |   |   |   Correlated Subquery.
|   |   |   |   Subquery under an IN predicate.
|   |   |   |
|   |   |   |   |SCALAR AGGREGATE Operator (VA = 3)
|   |   |   |   |  Evaluate Ungrouped ANY AGGREGATE.
|   |   |   |   |  Scanning only up to the first qualifying row.
|   |   |   |   |
|   |   |   |   |   |SCAN Operator (VA = 2)
|   |   |   |   |   |  FROM TABLE
|   |   |   |   |   |  titleauthor
|   |   |   |   |   |  Using Clustered Index.
|   |   |   |   |   |  Index : taind
|   |   |   |   |   |  Forward Scan.
|   |   |   |   |   |  Positioning by key.
|   |   |   |   |   |  Keys are:
|   |   |   |   |   |    au_id ASC
|   |   |   |   |   |  Using I/O Size 4 Kbytes for data pages.
|   |   |   |   |   |  With LRU Buffer Replacement Strategy for data pages.
|   |   |   |
|   |   |   |  END OF QUERY PLAN FOR SUBQUERY 1.
|   |   |
|   |   |  |SCAN Operator (VA = 5)
|   |   |  |  FROM TABLE
|   |   |  |  authors
|   |   |  |  Table Scan.
|   |   |  |  Forward Scan.
|   |   |  |  Positioning at start of table.
|   |   |  |  Using I/O Size 4 Kbytes for data pages.
|   |   |  |  With LRU Buffer Replacement Strategy for data pages.

Total estimated I/O cost for statement 2 (at line 1): 1.

 -----------
           0

(1 row affected)
```

Given the lack of search parameters on authors and the indexing on titleauthor in the pubs2 example database, the tablescans in the above are not surprising. The point is, however, that unlike show_cached_text() whose results are part of the result set, the show_plan() function output is in a typical "print" format. Consequently, it is best often executed using a single SSQLID as parameter vs. embedded in a query the way show_cached_text() was illustrated earlier. Other statement cache controls include:

dbcc purgesqlcache – removes all cached statements

dbcc purgesqlcache(SSQLID) - removes the particular cached statement by the SSQLID

dbcc prsqlcache – prints summaries of the statements in the statement cache

set statement cache [on | off] – session level command to enable or disable statement caching as well as using cached statements for execution.

While you can purge the statement cache, you need to first make sure that you have resolved the problem that resulted in a bad query plan being cached to begin with. Otherwise, at the next likely re-execution of the query, once again, a bad plan will be cached and reused.

For applications that submit a wide variety of statements with 'LIKE' clauses, range queries or other constructs that could result in bad query executions, one option would be to include 'set statement_cache off' in a login trigger for those applications. This could not only help these applications by not invoking bad query plans (or causing bad ones to be used by OLTP applications), it also could reduce some of the statement cache volatility.

Statistics Cache Usage

One of the biggest consumers of procedure cache is the index histograms used during optimization. This is actually a two-fold problem. First, the large consumption of procedure cache by index histograms can push stored procedures out of the procedure cache – or depending on the query complexity, prevent loading of other column histograms. The second problem is that if the procedure cache is too small, often these statistics have to be re-loaded from the sysstatistics system table – which is subject to the normal data caching constraints as is any table. At many customers, the end result is that query optimization time dramatically increases due to the amount of physical I/O's necessary to reload index or column histograms. A solution to this problem will be discussed more in the next section.

Monitoring Statistics Proc Cache Usage

For now, you can use the following MDA tables to try to get a picture of how much procedure cache is in use – not only for statistics – but all the other uses as well.

```
create existing table monProcedureCacheMemoryUsage (
        InstanceID                      tinyint,
        AllocatorID                     int,
        ModuleID                        int,
        Active                          int,
        HWM                             int,
        ChunkHWM                        int,
        NumReuseCaused                  int,
        AllocatorName                   varchar(30) NULL,
)
materialized
at "$monProcedureCacheMemoryUsage"
go
grant select on monProcedureCacheMemoryUsage to mon_role
go

/*
** monProcedureCacheModuleUsage definition
*/
if (exists (select * from sysobjects
        where name = 'monProcedureCacheModuleUsage'
          and type = 'U'))
        drop table monProcedureCacheModuleUsage
go
print "Creating monProcedureCacheModuleUsage"
go
create existing table monProcedureCacheModuleUsage (
        InstanceID                      tinyint,
        ModuleID                        int,
        Active                          int,
        HWM                             int,
        NumPagesReused                  int,
        ModuleName                      varchar(30) NULL,
)
materialized
at "$monProcedureCacheModuleUsage"
go
grant select on monProcedureCacheModuleUsage to mon_role
go
```

The last one – monProcedureCacheModuleUsage is often a good place to start as it provides high level procedure cache usage. Consider the following results from an idle server:

InstanceID	ModuleID	Active	HWM	NumPages Reused	ModuleName
0	1	2	32	0	Parser
0	2	0	10	0	Utilities
0	3	0	0	0	Diagnostics
0	4	0	120	0	Optimizer
0	5	31	532	0	Execution
0	6	0	14	0	Access
0	7	0	6	0	Backup
0	8	0	34	0	Recovery
0	9	0	2	0	Replication
0	10	791	1186	0	Procedural Objects
0	11	0	5	0	Sort
0	12	2	2	0	HK GC
0	13	0	0	0	HK Chores
0	14	0	0	0	BLOB Management
0	15	9	9	0	Partition Conditions
0	16	1	1	0	Pdes Local HashTab
0	17	123	133	0	Statement Cache
0	18	2	2	0	CIS
0	19	1	2	0	Frame Management
0	20	0	0	0	AuxBuf Management
0	21	0	1	0	Network
0	22	66	66	0	Procmem Control
0	23	1	1	0	Data change
0	24	0	0	0	Dynamic SQL
0	25	0	0	0	Cluster Threshold Manager
0	26	1	1	0	Multiple Temporary Database
0	27	0	0	0	Workload Manager
0	28	0	0	0	Transaction

As you can see, the biggest consumers of proc cache in the above example are:

- Procedural Objects (Stored Procedures)
- Statement Cache
- Execution
- Optimizer

Sort & Utilities are highlighted above as well as they may commonly appear at different times to be higher than expected. For example, if running 'CREATE INDEX' or 'UPDATE STATISTICS', both of these commands will impact not only the sort module, but also the utilities module. Certain queries that do in-memory sorts (such as merge joins) may also cause proc cache usage in this area.

Sometimes it is useful to plot the different modules over time to get a sense of the high water marks in relationship with each other to better grasp the maximum procedure cache usage. Consider the following graph from a customer production system showing predominantly stored procedures and query execution as leading consumers of procedure cache during daytime hours:

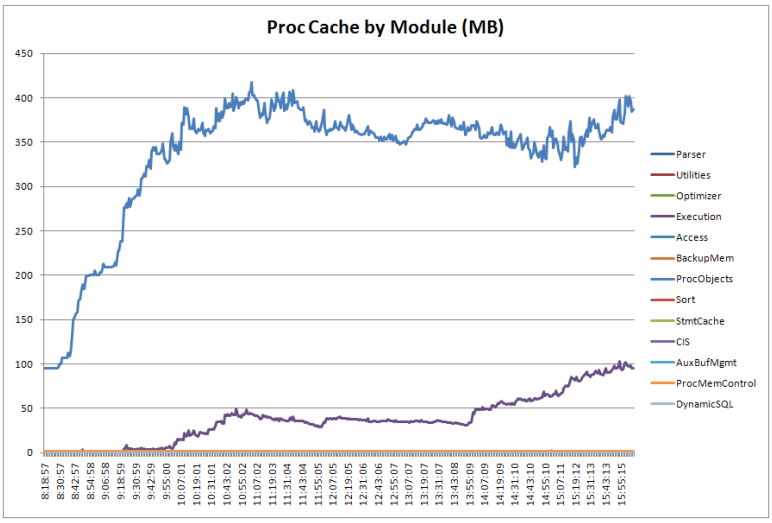


Figure 20 – Example Plot of Procedure Cache Usage by Module

Although the HWM for procedures object was the highest at 11am, the maximum procedure cache usage was at 15:55 when the Execution HWM in combination with the high procedure object usage raised the procedure cache total utilization to ~500MB.

To get a deeper understanding of exactly where the memory is used within these modules, we need to look on monProcedureCacheMemoryUsage. The below snapshot is from the same system as the earlier system used for the table on monProcedureCacheModuleUsage. Note the ModuleID correspondence with the ModuleID in the previous example:

InstanceID	AllocatorID	ModuleID	Active	HWM	Chunk HWM	NumReuse Caused	AllocatorName
0	110	1	2	2	1	0	MEMC_PARSLEX_1
0	112	1	0	2	1	0	MEMC_SQLPARS_2
0	113	1	0	30	2	0	MEMC_CURSOR_1
0	28	2	0	1	1	0	MEMC_LEGALINX_1
0	44	2	0	10	2	0	MEMC_STATS_4
0	231	2	0	2	1	0	MEMC_CRDB
0	115	4	0	118	2	0	MEMC_SCOMPILE_2
0	140	4	0	2	1	0	MEMC_OPTGLOBAL_1
0	7	5	0	4	1	0	MEMC_CONNHDLR_1
0	8	5	1	2	1	0	MEMC_CONNHDLR_2
0	9	5	0	1	1	0	MEMC_DATSERV_1
0	11	5	0	1	1	0	MEMC_ESP_1
0	114	5	933	948	2	0	MEMC_SCOMPILE_1
0	116	5	10	10	8	0	MEMC_EXECUTE_1
0	119	5	0	1	1	0	MEMC_SEQUENCER_2
0	134	5	0	4	1	0	MEMC_LECURSOP_1
0	135	5	0	6	2	0	MEMC_LEHASHOP_1
0	136	5	0	10	1	0	MEMC_LEHASHOP_2
0	164	5	0	1	1	0	MEMC_SEQLOCK_1
0	46	6	0	14	2	0	MEMC_INTSQL_1
0	155	7	0	6	6	0	MEMC_DUMPDB_1
0	35	8	0	1	1	0	MEMC_RECOVERY_1
0	36	8	0	1	1	0	MEMC_RECOVERY_2
0	38	8	0	1	1	0	MEMC_RECOVERY_4
0	39	8	0	1	1	0	MEMC_RECOVERY_5
0	40	8	0	2	2	0	MEMC_RECOVERY_6
0	48	8	0	9	8	0	MEMC_XITEM_1
0	49	8	0	1	1	0	MEMC_XITEM_2

InstanceID	AllocatorID	ModuleID	Active	HWM	Chunk HWM	NumReuse Caused	AllocatorName
0	106	8	0	9	8	0	MEMC_PFTS_1
0	210	8	0	9	8	0	MEMC_XITEM_3
0	98	9	0	2	1	0	MEMC_RABIFN_1
0	90	10	0	658	2	0	MEMC_PROCM_1
0	91	10	0	6	1	0	MEMC_PROCM_2
0	93	10	0	1	1	0	MEMC_PROCM_4
0	101	11	1	1	1	0	MEMC_SOLM_1
0	197	11	0	4	1	0	MEMC_LEHASHOP_5
0	198	11	4	4	1	0	MEMC_LESORTOP_1
0	96	12	2	2	1	0	MEMC_DOLHKGC_2
0	107	15	9	9	8	0	MEMC_PTNCOND_1
0	181	16	1	1	1	0	MEMC_PDESLHASH_1
0	126	17	19	19	9	0	MEMC_SQTCACHE_3
0	127	17	0	22	2	0	MEMC_STMTCACHE_1
0	142	18	2	2	1	0	MEMC_OMNIREM_1
0	162	19	1	1	1	0	MEMC_FRMMGR_1
0	163	19	0	1	1	0	MEMC_FRMMGR_2
0	10	21	0	1	1	0	MEMC_SITE_1
0	178	22	66	66	33	0	MEMC_PMCTRL_1
0	179	23	1	1	1	0	MEMC_DATACHG_1
0	214	26	1	1	1	0	MEMC_MULTEMPDB_1

As illustrated, for the Optimizer and Execution modules, almost all the procedure cache usage was in the area of MEMC_SCOMPILE_1 and MEMC_SCOMPILE_2 – allocations used during compiling statements. In addition, most of the memory used for stored procedures used small page (1 page) allocations instead of 2 or 4 page allocations – which could indicate that the cached stored procedures are fairly small.

Histogram Steps

To get a better idea of which tables/columns may be filling proc cache, you can run a query such as the below in each user database – focusing on the *RequestedSteps* and *ActualSteps* result columns.

```
select TableName=object_name(ss.id),
        -- need to use strtobin() due to byte swapping on big endian
        ColumnName=col_name(ss.id,hextoint(strtobin(substring(ss.colidarray,1,2)))),
        Row_Count=st.rowcnt,
        RequestedSteps=convert(int,ss.c5),
        ActualSteps=convert(int,ss.c4),
        ApproxDistincts=convert(int,round(1/convert(double precision,ss.c15),0)),
        DistinctsPerStep=round(convert(int,round(1/convert(double precision,ss.c15),0))
                               /convert(float,convert(int,ss.c4)),0),
        Uniqueness=str(1.0/(convert(double precision,ss.c15)*st.rowcnt),12,10),
        RangeDensity=str(round(convert(double precision,ss.c2),10),12,10),
        TotalDensity=str(round(convert(double precision,ss.c3),10),12,10),
        UpdStatsDate=convert(varchar(20),moddate,100),
        DaysAgo=datediff(dd,moddate,getdate())
from sysstatistics ss, systabstats st
where ss.id > 100 and st.id > 100
  and ss.id=st.id
  and ss.formatid=100
  and st.indid in (0,1)
  and ss.c4 is not null
order by TableName, ColumnName
```

The key is to look for any column with a high "*ActualSteps*" – especially higher than 1,000. This can happen if the *RequestedSteps* is higher than 50 since the default 'histogram tuning factor' server configuration is 20. The result is that the actual histogram steps can be 20 times higher than requested – although unlikely. More likely, it will be 2-5 times larger. If such a column is found, the best thing to do is to first delete the statistics (using 'DELETE STATISTICS') and then

re-run 'UPDATE INDEX STATISTICS' but using a smaller step count in the 'using ### values' clause.

Keep in mind that often times we are using a single requested step count as a way to ease maintenance. The most accurate method of establishing the correct step count would be to consider each indexed column separate on the basis of how unique the values were in the table. In the above query, this value is returned for each column as 'ApproxDistincts' with a related value in 'DistinctsPerStep'. If the column is unique, realistically very few steps are needed. For example, identity columns or sequential key columns really only need 1 or 2 statistics at the most to define the range of values. If the column is not unique, but perfectly uniform distribution, again, only a few (10 or so) histogram steps would be necessary along with an overall uniqueness value that determines how many distinct values there are in the table. Of course, perfect uniform distribution is rare – and this is where features such as the 'histogram tuning factor' and requesting multiple steps become a bit more important. Consider the following table:

Data Characteristic	Distinct	Example	Number of Steps
Unique Columns	(all)	Identity, Sequential Key, GUID	Default (20)
High Cardinality	>65,000	Long varchar	The more distinct the fewer the steps needed - start with default and a few frequency cells for rare duplicates. Rely on histogram tuning factor for skew.
Medium Cardinality	>10,000 <65,000	Medium varchar(30), place names, last names, etc.	A medium number of range cells with decent frequency cells - 100-300 actual steps. Rely on histogram tuning factor for high and very high frequency cells and the 100-300 step range for medium frequency cells.
Low - Medium Cardinality	>1000 <10,000	Medium varchar(30), first names, etc.	200-500 with high frequency cells
Low Cardinality	>250 <1000	Short varchars	Default with frequency cells
Very Low – Low Cardinality	1-500	Lookup codes (product codes, event statuses, etc.)	One step per value. Since skew is inherent, histogram tuning ideally should end up
Date/Datetime	Most	Trade date, Event date, etc.	1 cell per range query span (i.e. week) with 2 cells per range as the max. Histogram tuning likely not a factor.

Notice that the lower the cardinality of the data, the more steps we need. This is where the number of distinct values vs. the number of steps is a consideration. For example, it is obvious that if we had an index on a Boolean such as 'true/false' we would expect at least 2 steps – one for each value. The same would be true if we only had 20 distinct values (e.g. product codes, event statuses) – the best statistics would be to have a separate histogram step for each value.

Obviously, this suggests that rather than running update statistics at a table level, we should run it at a column level using a technique similar to:

```
-- obtain table density stats, column density stats for index keys
-- and leading column statistics
update statistics <tablename> [using ### values]
go

-- run this to obtain individual column stats on each non-leading index column
update statistics <tablename> (<nonleading_column_name) using <###> values
```

The problem is that this can be quite an administrative effort to set up – especially on large schemas. Keeping track of the different numbers of histogram steps to use would be quite a task as well.

There is a third alternative – do the individual column stats technique, but only on the key large transaction tables. Consider the following example (based on RS performance statistics) using the results of the above query:

TableName	ColumnName	Row Count	Requested Steps	Actual Steps	Approx Distincts	Distincts Per Step
rs_config	objid	219	50	39	22	1
rs_config	optionname	219	50	50	160	3
rs_databases	dbid	34	50	46	34	1
rs_databases	dbname	34	50	22	11	1
rs_databases	dsname	34	50	20	10	1
rs_databases	ldbid	34	50	44	31	1
rs_databases	ltype	34	50	4	2	1
rs_databases	ptype	34	50	6	3	1
rs_routes	dest_rsid	2	50	4	2	1
rs_routes	source_rsid	2	50	4	2	1
rs_sites	id	2	50	4	2	1
rs_sites	name	2	50	4	2	1
rs_statcounters	counter_id	388	50	50	388	8
rs_statdetail	counter_id	7947973	50	84	352	4
rs_statdetail	instance_id	7947973	50	131	999	8
rs_statdetail	instance_val	7947973	50	407	738	2
rs_statdetail	run_id	7947973	50	362	361	1
rs_statrun	run_date	361	50	50	361	7
rs_statrun	run_id	361	50	50	361	7

For those unfamiliar with RS monitor counter data, the statistics details are kept in the highlighted table – rs_statdetail. The primary key on the table is the combination of the columns {run_id, instance_id, instance_val, counter_id}. These same columns appear in multiple indexes to speed query performance. There also is a parent-child relationship between rs_statrun and rs_statdetail, and a foreign key lookup relationship between rs_statcounters and rs_statdetail on counter_id. Finally, there is an optional relationship between rs_statdetail.instance_id and rs_databases.dbid or ldbid as well as rs_sites.id.

Considering this, look at counter_id. In the parent lookup table, there are 388 distinct values. Of these 388, 352 of the values are used in rs_statdetail across the nearly 8 million rows. However, only 84 histogram steps are used with an average of 4 counter_id values per step. By contrast, consider run_id. Run_id is the unique key to the parent table rs_statrun. As can be seen, there were 361 samples of statistics taken. In looking at the histogram steps for rs_statdetail, notice however, that in rs_statdetail it uses 361 distinct values. This appears to be odd considering that both have nearly identical distinct values. However, in looking at the distinct values, the 352 counter_id's have only 13 different row counts while the 361 run_id's have 218 different row counts. In other words, the counter_id data is much more evenly or

uniformly distributed. Is this an accurate assessment? No. But, remember, the optimizer doesn't understand how the data is being used and is trying to balance query optimization with resource usage. The point is that truly optimal statistics may require understanding how the data is used. This requires looking at two perspectives:

- Database design perspective – Understanding which columns are higher cardinality lookup codes (but still low cardinality overall) such as the counter_id example above
- Query perspective – understanding which columns are often used in range queries such as described for date/time/datetime columns in the above table

If you run update statistics on a per table basis, you may need to re-run update statistics on those columns afterwards – *if query performance problems were noted due to index selection*. For example, if we learned that certain queries with counter_id as one of the predicates was not picking the index as it should, then running update statistics on that column might be advisable. Knowing we have ~388 distinct values, we could try the following command.

```
update statistics rs_statdetail (counter_id) using 400 values
```

The result is counter_id now has 391 actual steps. If we use 1000 steps, we get 452 actual steps. Remember, for very low cardinality cases (e.g. true/false) we should expect up to 2 times the number of distinct values as the frequency cells will be interspersed with range cells with a 0 value. When the number of steps exceeds the number of distinct values but is less than twice the number of cells, then some of the discrete values are still using range cells vs. frequency cells. Consider the following first several rows of the optdiag output after requesting 1000 steps (452 actual steps as result):

```
Histogram for column:                  "counter_id"
Column datatype:                       integer
Requested step count:                  1000
Actual step count:                     452
Sampling Percent:                      0
Out of range Histogram Adjustment is DEFAULT.

     Step     Weight                    Value

        1     0.00000000       <=       3999
        2     0.00013626       <=       4004
        3     0.00013626       <=       4007
        4     0.00013614       <=       4017
        5     0.00004542       <=       4018
        6     0.00000000        <       5000
        7     0.00095421        =       5000
        8     0.00000000        <       5002
        9     0.00095421        =       5002
       10     0.00095421       <=       5003
       11     0.00000000        <       5005
       12     0.00095421        =       5005
       13     0.00000000        <       5007
       14     0.00095421        =       5007
       15     0.00000000        <       5009
       16     0.00095421        =       5009
       17     0.00000000        <       5011
       18     0.00095421        =       5011
       19     0.00000000        <       5019
       20     0.00095421        =       5019
       21     0.00095421       <=       5020
       22     0.00000000        <       5022
       23     0.00095421        =       5022
       24     0.00000000        <       5024
```

Counter_id=5002 is using a typical frequency cell at steps 8 & 9. Step 10 looks like a range cell – and it is, but in this case it is a range of a single value (counter_id is an int datatype). In one sense, a frequency cell wouldn't be necessary, but sloppy query coding where a predicate

expression such as counter_id=5002.5 could result in a bad possible estimate for rows being returned. However, it is impossible to force the update statistics command to do precisely as we wish in this case. Turning off the histogram tuning by setting it to 1 eliminates all of the frequency cells and results in 279 actual steps even when requesting 1000. So the final technique is:

- run update statistics on the table using a reasonable number of steps
- run update statistics on the low cardinality lookup keys using a high requested step count
- run update statistics on other foreign keys with low to medium cardinality using double the number of requested steps if desired.
- run update statistics on date/time columns or other range query centric columns using the 1-2 cells per expected range as an estimate
- run update statistics on all other columns using a reasonable number of steps

When done, ensure that the number of histogram steps for any one column does not exceed 500-1000 steps or else you might end up with considerable proc cache consumption during optimization. If any columns do exceed the limit, use a command sequence similar to the following with a reduced step count.

```
delete statistics <tablename> (<columnname>)
go
update statistics <tablename> (<columnname>) using <###> values
go
```

Failure to first delete the statistics may cause the subsequent update statistics to inherit the previous step count.

How do the techniques compare? Consider the following schema/index on rs_statdetail:

```
create table rs_statdetail (
   run_id               rs_id                          not null,
   instance_id          int                            not null,
   instance_val         int                            not null,
   counter_id           int                            not null,
   counter_obs          numeric(22,0)                  null,
   counter_total        numeric(22,0)                  null,
   counter_last         numeric(22,0)                  null,
   counter_max          numeric(22,0)                  null,
   label                varchar(255)                   null,
   constraint PK_RS_STATDETAIL primary key (run_id, instance_id, instance_val, counter_id)
)
lock datarows
go
create nonclustered index counter_idx
        on rs_statdetail(counter_id)
go

create nonclustered index instance_counter_idx
        on rs_statdetail(instance_id, counter_id)
go
```

Then let's compare the two techniques:

```
set statistics time on
go
delete statistics rs_statdetail
go
-- clear the cache by binding and unbinding the table
exec sp_bindcache tempdb_cache, rep_analysis_152, rs_statdetail
go
exec sp_unbindcache rep_analysis_152, rs_statdetail
go
```

```
-- technique #1 – normal update index stats on full table
update index statistics rs_statdetail using 50 values
go

delete statistics rs_statdetail
go
-- clear the cache by binding and unbinding the table
exec sp_bindcache tempdb_cache, rep_analysis_152, rs_statdetail
go
exec sp_unbindcache rep_analysis_152, rs_statdetail
go



-- technique #2 – update statistics followed by individual columns
-- for the possible low cardinality keys, we will use a high step count
-- for other columns of less concern, we will use our default 50
-- requested steps
update statistics rs_statdetail using 50 values
go
update statistics rs_statdetail (counter_id) using 1000 values
go
-- using more than 500 steps causes us to exceed our quota since we have
-- more than 500 distincts.  We will just double our default request though.
update statistics rs_statdetail (instance_id) using 100 values
go
update statistics rs_statdetail (instance_val) using 50 values
go
```

The results:

Metric	Update index statistics	Update statistics/ update col statistics
CPU Time (ms)	23000	Upd stats = 5800 Counter_id = 10800 Instance_id = 11500 Instance_val = 16300 TOTAL = 44400
Elapsed Time (ms)	149553	Upd stats = 44510 Counter_id = 10740 Instance_id = 11533 Instance_val = 82440 TOTAL = 149223
Run_id steps	362	362
Counter_id steps	84	452
Instance_id steps	131	628
Instance_val steps	407	407

As noticed, we get more histogram steps yet consume the same amount of elapsed time but double the CPU time due to the extra sorts since we have both counter_id and instance_id as leading columns of indexes already. If we would have just updated statistics on non-leading columns (instance_val in this case) we would have used the same CPU time and finished 14% quicker.

After doing this we will need to both monitor procedure cache as well as the query plans to see if it resolved the query issues without consuming too much additional procedure cache. Once this is stable, the key to ensuring query optimization is not impacted is always ensure that systabstats and sysstatistics are in cache.

Recommended Named Caches

There is multiple ways that you can arrange the data cache – and each have their benefits. So, it could be said there are multiple "right" ways to configure the data cache. It also could be said that there are multiple "wrong" ways as well. A lot of what is "right" vs. "wrong" is often application dependent, consequently, relying on web gouge or standard implementations for

every situation is likely to cause problems. For example, in today's oft situation of server consolidation where the server is simultaneously hosting different applications – is it "right" or "wrong" to consider an "application"-oriented cache configuration vs. a broader server/object-oriented approach? The "right" answer depends on the application SLA's, the degree of separation necessary to meet those SLA's – and the extent of application isolation (vs. shared data).

Having said that, there are a couple of generic recommendations about named caches that should be considered as they apply to most typical situations. These recommendations, outlined in the below paragraphs, represent a starting point that would need to be tuned or adapted for each server's specific requirements.

Log Cache

Primary application databases should have their transaction logs bound to a dedicated log cache. In server consolidation situations with 10's to 100's of databases, a secondary or tertiary log cache should also be considered for auxiliary processing databases.

ULC Flushes & Write Performance

Database performance typically benefits quite a bit from a log cache. Reflecting back on the discussion of MASS writes and large IO's, during the typical log writing, an user's modified records are flushed from their Private Log Cache (aka User Log Cache) to the log cache in a ULC-sized chunk. As a result, it makes sense then to realize that:

- The ULC size needs to be large enough to hold most of a single transaction for most processing.
- The log cache needs have most of the cache configured in a pool the same size as the ULC
- The log's IO size (sp_logiosize) needs to be set to the same as the pool/ULC size

Most people are familiar with this section of the sp_sysmon output:

```
Transaction Management
----------------------

  ULC Flushes to Xact Log         per sec      per xact       count  % of total
  -------------------------  ------------  ------------  ----------  ----------
    by Full ULC                     65.1           8.0       19581      33.7 %
    by End Transaction               8.5           1.0        2565       4.4 %
    by Change of Database            0.0           0.0           5       0.0 %
    by Single Log Record             0.6           0.1         184       0.3 %
    by Unpin                       119.0          14.5       35810      61.6 %
    by Other                         0.0           0.0           0       0.0 %
  -------------------------  ------------  ------------  ----------
  Total ULC Flushes                193.2          23.6       58145

  ULC Flushes Skipped             per sec      per xact       count  % of total
  -------------------------  ------------  ------------  ----------  ----------
    by PLC Discards                  0.0           0.0          14     100.0 %
  -------------------------  ------------  ------------  ----------
  Total ULC Discards                 0.0           0.0          14

  ULC Log Records                 8119.0         992.6     2443830       n/a
  Max ULC Size During Sample         n/a           n/a           0       n/a

  ULC Semaphore Requests
    Granted                      16340.0        1997.7     4918329      99.8 %
    Waited                          25.8           3.2        7771       0.2 %
  -------------------------  ------------  ------------  ----------
```

```
Total ULC Semaphore Req         16365.8        2000.9     4926100

Log Semaphore Requests
  Granted                         204.2          25.0       61455      97.4 %
  Waited                            5.4           0.7        1622       2.6 %
-------------------------  ------------  ------------  -----------
Total Log Semaphore Req           209.6          25.6       63077

Transaction Log Writes             84.4          10.3       25391       n/a
Transaction Log Alloc              80.7           9.9       24279       n/a
Avg # Writes per Log Page           n/a           n/a     1.04580       n/a

Tuning Recommendations for Transaction Management
-------------------------------------------------
- Consider increasing the 'user log cache size'
  configuration parameter.
```

Similar information at a more discrete user level (vs. aggregated server level statistics) can be obtained from the MDA tables. For example, the above suggests that the DBA should increase the 'user log cache size' as non-system related ULC flushes were tilted heavily towards 'Full ULC' and 'End Transaction'. However, if the system is supporting a mix of OLTP and batch processes, the batch processes are likely to fill the ULC no matter how large it is created (reasonably). Secondly, considering the number of ULC Flushes by Unpin, increasing the size may not help if the predominant OLTP transactions are affecting datarows tables with high concurrency causing ULC flush. Consequently, it might be good to get an average from key OLTP processes and specifically excluding batch processes by reviewing the MDA data across the user classes:

```
create existing table monProcessActivity (
        SPID                            int,
        InstanceID                      tinyint,
        KPID                            int,
        ServerUserID                    int,
        CPUTime                         int,
        WaitTime                        int,
        PhysicalReads                   int,
        LogicalReads                    int,
        PagesRead                       int,
        PhysicalWrites                  int,
        PagesWritten                    int,
        MemUsageKB                      int,
        LocksHeld                       int,
        TableAccesses                   int,
        IndexAccesses                   int,
        TempDbObjects                   int,
        WorkTables                      int,
        ULCBytesWritten                 int,
        ULCFlushes                      int,
        ULCFlushFull                    int,
        ULCMaxUsage                     int,
        ULCCurrentUsage                 int,
        Transactions                    int,
        Commits                         int,
        Rollbacks                       int,
)
materialized
at "$monProcessActivity"
go
```

Ideally, core OLTP processes should have the ULCBytesWritten/ULCFlushes and ULCMaxUsage both fairly close the ULC size One interesting statistic is that both ULCBytesWritten and ULCMaxUsage will be the sum of both the session tempdb ULC and the normal SPID ULC, consequently it may be difficult to determine the true maximum of the normal ULC. Equally unfortunate, the MDA tables do not track all the reasons why ULC flushes happen. So without sp_sysmon, it can be difficult to ascertain when the ULC is simply sized too large or not effective due to other reasons. Two of the most common reasons the ULC may be flushed when not full and prior to the end of a transaction is due to a Single Log Record (SLR) (in 12.5.x and earlier) or Buffer Unpinning (all ASE versions since 11.9.2).

ULC Flush: Buffer Unpinning

"Buffer Unpinning" is perhaps the best one to start with. Keep in mind that this only is concerned with datarows locking as it involves situations in which two different user sessions modify the same page during concurrent transactions – so it will not happen for datapage or allpages locked tables. When a user transaction modifies a data row, the data page image in memory is changed immediately. At this point, since the transaction has not yet committed, the lock is maintained on that row. However, a different user could modify a different row on that page. If a user's transaction is rolled back, that user's changes to the page need to be undone and the previous page image restored. Pinning the datapage buffer to the ULC achieves several aspects:

- The housekeeper and checkpoint can skip writing that page as the transaction is not committed and would only need to be rolled back on recovery.
- The process of rolling back the transaction is faster as the ULC contains the before/after images of the rows modified
- Datapage location in memory is faster vs. searching for them (during rollbacks) as the locations of the pinned pages in memory are tied to the ULC.

The last point is fairly obvious, so we will skip discussing it. However, the first aspect explains why a previous user's ULC needs to be flushed when a subsequent user modifies a different row on the page. If the buffer is pinned to the subsequent user's ULC and the housekeeper/checkpoint are not going to write it to disk, then the only way to be sure there is a permanent record of a user's changes to commit or rollback (or heuristically complete for XA) is if the transaction log contains the rows. This may sound a bit strange as the oft asked question is why can't it still wait until the first user's transaction commits to flush the ULC anyhow. The answer is that it isn't the after images that are important – but the before images. During recover, remember, that the server needs to rollback any uncommitted transactions. Consider the following sequence:

	User 1	User 2	User 3
T1	Begin tran Update row1 page1		
T2		Begin tran Update row2 page1 Commit tran	
T3			Begin tran Update row3 page1

What is on disk is likely the page at T0 – before the rows were modified. Consider what would happen without buffer unpinning/ULC flushing if the system crashed at T4:

- At end of T2, with no buffer pinning, a disk write could have happened to flush the page to disk. This write would have included the committed row2 and uncommitted row 1 row images.
- At end of T3, with the page pinned to user3, no disk write would have happened. During recovery, this would reduce the amount of work to "undo".
- At recovery, without T1's log records in the transaction log, there would be no way to "undo" the uncommitted row modification to row 1

Consequently, whenever a subsequent transaction modifies the same page as a current transaction, the previous transaction's log records are flushed to disk. So in the above sequence, User2 would have forced a ULC Flush by Unpin on User1, User2 would have had a ULC Flush by End Transaction, and User3 would not have had a ULC Flush yet by the time the system crashed. Note that User2 would have released the buffer pinning when the transaction committed in order to allow checkpoint or housekeeper to process the pages if necessary. User 1 would have re-pinned the pages.

As far as the second aspect – faster rollbacks due to row image availability, consider the scenario of a user that does a bulk update of 10,000 rows with a series of statements vs. a smaller transaction, such as:

```
-- user 1
begin tran
update table set status='90 days overdue'
  where DueDate < dateadd(dd,-90,getdate())
    and DateRecv is null
update table set status='60 days overdue'
  where DueDate < dateadd(dd,-60,getdate())
    and DateRecv is null
update table set status='30 days overdue'
  where DueDate < dateadd(dd,-30,getdate())
    and DateRecv is null
update table set status='delinquent'
  where InCollections=1

-- user 2
update table set DateRecv=getdate() where InvoiceID=123456
```

Now, at this point, consider what happens if a network glitch bumps both users offline and their transactions need to be rolled back. In the first case, 10's of thousands of records were likely modified – and very likely all the modifications didn't fit in the ULC. Secondly, the last update statement might have impacted some of the same records that were modified in the first one or two statements. Consequently, to rollback the transaction and undo all the row modifications to restore the datapage to its original unaltered state, ASE will need to do a log scan to find the original row images. In the second case, since the before/after images are in the ULC and ASE knows this because the page is pinned to that ULC, it simply can use the ULC images vs. a log scan.

This should also make clear why buffer unpinning is not used for ULC flushes for IMDB or Reduced Durability Databases (minimally logged). Since recovery is not guaranteed – nor expected – in those cases, the log records are not flushed to the transaction log when the buffers are unpinned. In the case of the RDDB, if a normal shutdown occurs, of course, the final checkpoint will flush the final (and committed) pages to disk to ensure a consistent database image on recovery. If a unexpected shutdown happens, however, the pages on disk would likely have uncommitted row modifications (User1 in the earlier scenario) but with no way to back it out as there would be no log records to do so with. While uncommitted data is a minor concern, the bigger issue is any page linkages or allocation changes that also would have been lost – consequently, a RDDB simply restores from a template vs. trying to recover any data.

From a performance angle, too many ULC Flushes by Unpin may result in excessive log semaphore contention – slowing down processing. Consider the earlier example sp_sysmon:

```
ULC Flushes to Xact Log         per sec      per xact       count  % of total
-------------------------  ------------  ------------  ----------  ----------
  by Full ULC                      65.1           8.0       19581      33.7 %
  by End Transaction                8.5           1.0        2565       4.4 %
  by Change of Database             0.0           0.0           5       0.0 %
  by Single Log Record              0.6           0.1         184       0.3 %
```

```
  by Unpin                      119.0          14.5        35810      61.6 %
  by Other                        0.0           0.0            0       0.0 %
-------------------------  ------------  ------------  ----------
Total ULC Flushes               193.2          23.6        58145
```

From this we can easily conclude that there is a ton of concurrency – inserts, updates or deletes – all affecting the same pages. Looking at the log semaphore contention for the same sp_sysmon, we see:

```
Log Semaphore Requests
  Granted                       204.2          25.0        61455      97.4 %
  Waited                          5.4           0.7         1622       2.6 %
-------------------------  ------------  ------------  ----------
Total Log Semaphore Req         209.6          25.6        63077
```

In this case, the log semaphore waits are not too bad – 5/second or 2.6%. If it was much higher to where it was a leading cause of system degradation, we would have to determine why the concurrency. For example:

- Ascending, concurrent inserts into a heap table or table whose clustered index was on a monotonically increasing key (such as a current datetime field or identity column)
- Parallel updates from a parallel batch process crawling through a large table – or a job queue in which subsequent processes need to process the records in serial order.

If possible, sometimes this contention can be reduced by (a) reclustering the table on a different key to scatter the inserts; (b) assigning the batch processes ranges to work on to deconflict (and it would also reduce latch contention); or (c) reducing the max_rows_per_page to increase the pages involved. The first two are familiar to most long time DBA's as the techniques predated DOL locking in ASE. The last one may be a bit strange, however. Think of it this way: if 100 rows of a job queue were contained on 25 pages (4 rows per page) and 100 processes were working concurrently, there would have been 75 ULC Flushes by Unpin (3 per page) as each job locked its respective row – not to mention 3 latch waits per page. Now, consider if the max_rows_per_page was set to 1. Yes, it would take up 100 pages instead of 4, but there would have been no ULC Flushes by Unpin (and no latch contention). Yes, queries checking the number of outstanding jobs might incur more IO's and be slower now – but overall system throughput of job processing may be higher due to decreased log semaphore contention.

ULC Flush: System (Single) Log Record (SLR)

This topic is more for historical perspective as ASE 15.0.2+ changes have fairly much eliminated this particular issue from being a primary contributor. We are all aware that ASE logs much more than row modifications. For example DDL changes are recorded as are space allocations. In fact, we all know that the term 'minimally logged' refers to the fact that while row/page modifications are not logged, space allocations are. OAM pages represent one of several different system log records that can be in a user's ULC. If the object is shared (i.e. not a #temp), that space allocation has to be logged immediately for recovery reasons if the database has mixed log and data segments (such as tempdb does). One of the enhancements in 15.0.x+ was that SLR's would not cause a ULC flush in databases in which there was no guaranteed recovery – tempdb's, RDDB and IMDB. As a result, ULC Flushes by SLR typically are fairly rare overall as a result in 15.0 and higher. If you see a significant number of these, it may because of

transactions in master, sybsystemdb, sybsystemprocs – all of which are combined data/log on single device – or it may be because a database was not created correctly with separate data and log segments.

Log Dumps & Cache Sizing

On very high transaction rate systems without truncate log on checkpoint set, frequent transaction log dumps are required to prevent the log from filling and transactions suspending. However, if the size of the dump exceeds the log cache, then the dump transaction request must perform physical reads to scan the transaction log. This problem is especially prevalent in systems that do transaction log dumps every few minutes (10-15) or have it automated via log space thresholds and the thresholds are firing every few minutes.

In this case, the size of the log cache should be at least 25% larger than the largest log dump during peak operations. The reasons for considering peak operations is that it is most likely that if any problems develop in which a log dump cannot clear the space fast enough – it will most likely be during peak operation or off-peak batch processing. Obviously, peak processing is the more critical.

If the log dump is compressed, you may need to estimate the uncompressed log space by monitoring the log space usage as well as space freed after a dump. In theory, the only records remaining in the transaction log after a log dump are those that occur after the oldest open transaction. Records before that point are known to be committed and can be safely be removed. Consequently, the space freed from the log plus the space still consumed in the log represent the amount of log space – and hence log cache required.

Note that this consideration is only necessary when using frequent log dumps in a continuously high volume transaction system.

Log Scans & Cache Sizing

Other than enabling large IO's and better efficiency for ULC Flushes during writing, the biggest use of the log cache actually comes from log scans. As was alluded to earlier, this can happen during transaction rollbacks, pre-commit processing (for example certain deferred operations first log the rows and then use a log scan to identify the actual rows to modify), post-commit processing, triggers (to create the inserted/deleted tables), log based replication (ASE RepAgents) and of course checkpoints. Obviously, if any of these operations needs to a physical IO, the performance can suffer. The trick is to size the transaction log cache large enough to avoid this without wasting too much memory.

While DBA's are typically good about configuring a dedicated log cache, unfortunately, most grossly oversize the cache size and as a result, detract from memory that could be given to other named caches. In reality, the size of a named cache for transaction logs can be very small – as is noted in the Sybase ASE manuals:

> *On SMP systems with high transaction rates, bind the transaction log to its own cache to reduce cache spinlock contention in the default data cache. In many cases, the log cache can be very small.*
>
> *-- ASE 15.5 Performance & Tuning Series: Basics; Chapter 5: Memory Use and Performance; Named Cache Recommendations*

This can be proven by looking at sp_sysmon or MDA tables. Consider the earlier sysmon with:

```
  ULC Flushes to Xact Log         per sec      per xact       count  % of total
  -------------------------  ------------  ------------  ----------  ----------
…
```

```
  ULC Log Records                  8119.0         992.6     2443830       n/a
  Max ULC Size During Sample          n/a           n/a           0       n/a
…
  Transaction Log Writes             84.4          10.3       25391       n/a
  Transaction Log Alloc              80.7           9.9       24279       n/a
  Avg # Writes per Log Page           n/a           n/a     1.04580       n/a
```

This was from a trading system doing normal micro-batch inserts from a trade feed, so the number of log records and records per transaction are normal. The line to focus on, however, is the 'Transaction Log Alloc' line, which shows ~81 log pages being allocated every second. For a 5 minute recovery window checkpoint scan, we would estimate 81*60*5=24,300 pages – or 47MB. As you can see, actually quite small. Let's take a look at another sample from a 50+ engine server (30 minute sp_sysmon):

```
  ULC Flushes to Xact Log          per sec      per xact       count  % of total
  -------------------------  ------------  ------------  ----------  ----------
…
  ULC Log Records                22288.1           7.8    40408318       n/a
  Max ULC Size During Sample          n/a           n/a           0       n/a   …
  Transaction Log Writes          1020.0           0.4     1849241       n/a
  Transaction Log Alloc           1518.9           0.5     2753780       n/a
  Avg # Writes per Log Page           n/a           n/a     0.67153       n/a
```

Using the above logic, we might conclude that we would need a log cache of 1519*60*5=~900MB. However, there are a lot of factors to consider.

One factor in this consideration is that there were ~20 plus databases in this system. Not all of these may have the same recovery factor, transaction concurrency, use of triggers, etc. Secondly, some of these log writes include tempdb. This system used a lot of logged IO in tempdb by virtue of doing create table #temp followed by insert/selects to populate #temp as an attempt to try to reduce system table contention when running in ASE 12.5. Looking further down in sp_sysmon we can see a breakdown of IO by device such as:

```
  Device:
    /dev/vx/rdsk/sybaseASE/cap01log_fin
    cap01log                       per sec      per xact       count  % of total
  -------------------------  ------------  ------------  ----------  ----------
    Reads
      APF                              0.0           0.0           0       0.0 %
      Non-APF                          0.0           0.0           3       0.1 %
    Writes                             1.1           0.0        2030      99.9 %
  -------------------------  ------------  ------------  ----------  ----------
  Total I/Os                           1.1           0.0        2033       0.0 %

…
  Device:
    /dev/vx/rdsk/sybaseASE/data01log_fin
    data01log                      per sec      per xact       count  % of total
  -------------------------  ------------  ------------  ----------  ----------
    Reads
      APF                              0.0           0.0           0       0.0 %
      Non-APF                          0.1           0.0         197       0.3 %
    Writes                            36.3           0.0       65898      99.7 %
  -------------------------  ------------  ------------  ----------  ----------
  Total I/Os                          36.5           0.0       66095       0.9 %

…

  Device:
    /dev/vx/rdsk/sybaseASE/data02log_fin
    data02log                      per sec      per xact       count  % of total
  -------------------------  ------------  ------------  ----------  ----------
    Reads
      APF                              0.0           0.0           0       0.0 %
      Non-APF                          0.0           0.0           1       0.0 %
    Writes                             2.7           0.0        4876     100.0 %
  -------------------------  ------------  ------------  ----------  ----------
  Total I/Os                           2.7           0.0        4877       0.1 %
…
  Device:
    /dev/vx/rdsk/sybaseASE/data03log_fin
    data03log                      per sec      per xact       count  % of total
  -------------------------  ------------  ------------  ----------  ----------
```

```
  Reads
    APF                          0.0          0.0           0       0.0 %
    Non-APF                      0.1          0.0          99       0.5 %
  Writes                        10.6          0.0       19292      99.5 %
------------------------  ------------  ------------  ----------  ----------
Total I/Os                      10.7          0.0       19391       0.3 %
```

…and so on. After isolating all the logs, the real log writes for this system is only about 347 pages/sec – about 1/4th the 1,500/sec number anticipated. But still at 347 pages/sec, we would have a total of ~200MB need for the log cache.

In some cases, it might be easier to look at the disk writes for the log devices associated with the production databases using monIOQueue. For instance, to isolate the log writes above, it required manually reviewing hundreds of device statistics in a single sp_sysmon. For any sort of trending, this would have to be done dozens of times – a rather tedious task. Then, the accuracy depends of if the DBA followed the naming convention and only used devices with "log" in the name as log devices – or worse, due to sudden need to extend a log, extend a log on one of the "data" devices. The MDA table monIOQueue separates IO requests by the type of IO – log, data or tempdb – but has to be joined with monDeviceIO to separate reads and writes.

```
create existing table monIOQueue (
        InstanceID                      tinyint,
        IOs                             int,
        IOTime                          int,
        LogicalName                     varchar(30) NULL,
        IOType                          varchar(12) NULL,
)
materialized
at "$monIOQueue"

create existing table monDeviceIO (
        InstanceID                      tinyint,
        Reads                           int,
        APFReads                        int,
        Writes                          int,
        DevSemaphoreRequests            int,
        DevSemaphoreWaits               int,
        IOTime                          int,
        LogicalName                     varchar(30) NULL,
        PhysicalName                    varchar(128) NULL,
)
materialized
at "$monDeviceIO"
go
```

Consider the following output from monitoring a production system at a high volume package shipping company over 17h:47m:47s (note that some devices and rows appear twice due the join with monDeviceIO):

Logical Name	IOType	Reads	APFReads	Writes	Total IOs
data_device2	User Data	1,765,644	1,010,775	169,491	1,935,122
tempdb2	Tempdb Log	365,150	1,909	1,198,428	141,326
tempdb2	Tempdb Data	365,150	1,909	1,198,428	1,422,252
tempdb	Tempdb Log	142,500	8,437	339,189	0
tempdb	Tempdb Data	142,500	8,437	339,189	481,689
data_device3	User Data	17,824	10,863	2,905	20,729
data_device4	User Data	4,109	197	276	4,385
log_device2	User Log	1,795	0	543,259	539,718
log_device2	User Data	1,795	0	543,259	5,336
master	User Log	376	0	12,190	1,125

Log_device2 had a total of 539,718 IO's for 'User Log' and 5,336 for 'User Data' – which implies that some database likely mistakenly has a slight data segment extension on this device. With only 1759 Reads, the assumption is that the 'User Data' IO's represent some portion of the

"Writes" – but we don't know how much. Regardless, even if we assume that the total 539,259 writes over the period were all log, we could average that to ~30,000 pages per hour or only 500 pages per minute. A log cache of 2,500 pages or <5MB would suffice.

By contrast, it is not uncommon to see systems with configurations that allocate 100's of MB or even GB more memory to the log cache than necessary. One of the more common excuses for this is RepAgent latency. The reality is that the RepAgent latency has almost nothing to do with needing to physical reads of the log pages as most of the time is spent waiting on network interaction. Once configured, the size of the log cache can be reviewed via sp_sysmon. Consider the following snippet for the same 50+ engine server from above:

```
Cache: log_cache
                              per sec      per xact       count  % of total
-------------------------  ------------  ------------  ----------  ----------
  Spinlock Contention              n/a           n/a         n/a       0.0 %

  Utilization                      n/a           n/a         n/a       0.0 %

  Cache Searches
    Cache Hits                   462.3           0.2      838062     100.0 %
       Found in Wash               0.0           0.0           0       0.0 %
    Cache Misses                   0.0           0.0           0       0.0 %
-------------------------  ------------  ------------  ----------
  Total Cache Searches           462.3           0.2      838062

  Pool Turnover
    2  Kb Pool
         LRU Buffer Grab           0.7           0.0        1261       1.0 %
           Grabbed Dirty           0.0           0.0           0       0.0 %
    4  Kb Pool
         LRU Buffer Grab          66.2           0.0      119950      99.0 %
           Grabbed Dirty           0.0           0.0           0       0.0 %
-------------------------  ------------  ------------  ----------
  Total Cache Turnover            66.9           0.0      121211

  Buffer Wash Behavior
    Statistics Not Available - No Buffers Entered Wash Section Yet

  Cache Strategy
    Cached (LRU) Buffers          337.0           0.1      610968     100.0 %
    Discarded (MRU) Buffers         0.0           0.0           0       0.0 %

  Large I/O Usage
    Large I/Os Performed           66.2           0.0      119950      99.2 %

    Large I/Os Denied due to
      Pool < Prefetch Size          0.0           0.0           0       0.0 %
      Pages Requested
      Reside in Another
      Buffer Pool                   0.5           0.0         944       0.8 %
-------------------------  ------------  ------------  ----------
  Total Large I/O Requests        66.7           0.0      120894

  Large I/O Detail
   4   Kb Pool
      Pages Cached                132.3           0.0      239900       n/a
      Pages Used                    0.0           0.0           0       0.0 %

  Dirty Read Behavior
        Page Requests               0.2           0.0         312       n/a

  Tuning Recommendations for Data cache : log_cache
  -------------------------------------
  - Consider using 'relaxed LRU replacement policy'
    for this cache.
```

Note that even though we are using a 4K pool, we still have a 2K server. As a result, the pages cached are double the number of LRU buffer grabs. Based on our earlier estimates, we had anticipated ~200MB as an estimated size for the cache. In reality, what we see is we are caching 132 pages/second. For a 5 minute window, we then would need ~40,000 pages or only ~80MB – much less than estimated. Even if we doubled that, it would only be 150MB of cache necessary. However, you can see that during this same 30 minute sample, we actually cached

~240,000 pages or ~470MB. The reason why is that unlike a data cache, a log cache pages are not frequently re-read – the cache is simply appended to over time.

The net result is that for most medium sized systems a log cache of 50-100MB is fine, while extremely large SMP might need 200MB. The log cache should not be partitioned as the log semaphore protects from concurrency. If multiple databases reside in the same server, it might be wiser to consider 2-3 log caches with several logs per cache vs. a single larger cache.

One key recommendation for log caches is noted in the above sp_sysmon example output – *use a relaxed cache strategy*. As described in the documentation, user processes as well as system tasks such as the checkpoint do regularly read from the transaction log. When applications use triggers, deferred updates or transaction rollbacks, some log pages may be read if the transaction exceeded the size of the ULC. Typically, pages read by an application very recently used pages, which are still in the cache – and they are usually only read once – consequently moving them to the MRU end of the cache has little benefit. Similarly, the checkpoint or Replication Agent may be reading very old log pages from disk – pages that likely will never be accessed again.

System Table Cache

The need for a cache for ASE system tables seems almost unnecessary. Earlier, we talked about the metadata cache (DES, IDES, PDES), but the focus for those structures is for controlling object caching. During query compilation and optimization, ASE may still need to read from the system tables. As with anything, if you can avoid a physical read to achieve this, the speed is much better.

By default, most of the system tables would be cached in the default data cache. The problem is that system tables are not treated any more special than any other table. As cache turnover occurs, they will be bumped from cache – and then re-read over and over again. Key tables to think about:

syscolumns & sysobjects – query parsing and procedure deferred name resolution

sysconstraints – foreign key, check and other constraints to be enforced

sysprotects – user object permissions

sysprocedures – procedure tree for loading procs from disk/re-resolution

systabstats & sysstatistics – query optimization

sysmessages – error messages for stored procedures (if developers leveraged this feature as they should have – especially for localization)

Other tables may also be affected such as syscomments. The way to detect if cache turnover is affecting an existing system is to use monCachedObject and the following query:

```
create existing table monCachedObject (
        CacheID                         int,
        InstanceID                      tinyint,
        DBID                            int,
        IndexID                         int,
        PartitionID                     int,
        CachedKB                        int,
        CacheName                       varchar(30) NULL,
        ObjectID                        int NULL,
        DBName                          varchar(30) NULL,
```

```
        OwnerUserID                     int NULL,
        OwnerName                       varchar(30) NULL,
        ObjectName                      varchar(30) NULL,
        PartitionName                   varchar(30) NULL,
        ObjectType                      varchar(30) NULL,
        TotalSizeKB                     int NULL,
        ProcessesAccessing              int NULL,
)
materialized
at "$monCachedObject"


-- system tables all have an ObjectID < 100 currently
select CacheName, ObjectName, IndexID, CachedKB, TotalSizeKB, ProcessesAccessing
  from monCachedObject
  where ObjectID < 100
    and ObjectID !=8      -- ignore syslogs
    and (InstanceID is null or InstanceID = @@instanceid)
    and DBName not in ('sybsystemprocs','tempdb')   -- add user defined tempdb's to this list
```

If you see the 'CachedKB' fluctuating repeated (with the exception of tempdb system tables), you can be certain that you are experiencing physical reads during query parsing, optimization and possibly procedure execution.

Most experienced customers have created a separate named cache to contain the system tables. How large it needs to be depends on the schema size for all the databases within the system. One of the tricks to creating and using the system table cache is that in order to bind system tables to it, the database needs to be in single user mode – which requires an application outage. As a result, it can't be easily implemented or modified from a perspective of which tables to bind to it. Once created, it can be monitored using sp_sysmon like the others. Consider the following:

```
Cache: systables_cache
                                     per sec      per xact       count  % of total
-------------------------  ------------  ------------  ----------  ----------
  Spinlock Contention                 n/a           n/a         n/a       1.3 %

  Utilization                         n/a           n/a         n/a      38.4 %

  Cache Searches
    Cache Hits                  442433.2         454.6    13272996     100.0 %
      Found in Wash               4581.2           4.7      137435       1.0 %
    Cache Misses                     0.0           0.0           1       0.0 %
-------------------------  ------------  ------------  ----------
  Total Cache Searches          442433.2         454.6    13272997

  Pool Turnover                    0.0           0.0           0       n/a

  Buffer Wash Behavior
    Statistics Not Available - No Buffers Entered Wash Section Yet

  Cache Strategy
    Cached (LRU) Buffers        394851.2         405.7    11845535     100.0 %
    Discarded (MRU) Buffers          0.0           0.0           0       0.0 %

  Large I/O Usage
    Total Large I/O Requests         0.0           0.0           0       n/a

  Large I/O Detail
    No Large Pool(s) In This Cache

  Dirty Read Behavior
        Page Requests                0.0           0.0           0       n/a

  Tuning Recommendations for Data cache : systables_cache
  -----------------------------------
  - Consider using 'relaxed LRU replacement policy'
    for this cache.

  - Consider adding a large I/O pool for this cache.
```

The key is the high-lighted lines – which shows that 100% of the cache lookups in this case were satisfied from memory. A real interesting aspect is that for this system (a 64 engine ASE), 38.4%

of all cache lookups were related to system tables bound to this cache – so you can see the degree of impact that physical reads for system table data could have on your system.

Generally, due to the size of production schemas, the system table cache could be bigger than the log cache – likely in the 100MB-300MB range. Of particular note, since the cache hit rate is so high and there should be minimal writes – a system table cache is a great candidate for a relaxed cache strategy. However, unlike the advice above, a large I/O pool would likely not benefit. The reason is that if you created a large I/O pool of 25% of the cache size, system tables that might be able to use large I/O's (sysprocedures, sysstatistics, etc.) would only be able to use that much of the cache and you might get a lot of pool turnover (highlighted above). Cache partitioning may not be necessary – especially if using a relaxed cache strategy – as the pages should not be relocating and therefore spinlock contention should be minimal. The above shows slight contention, but then a relaxed cache strategy was not implemented.

Tempdb Cache(s)

The third "must create" cache is named caches for the tempdb's. In this case you may actually need multiple caches to help reduce spinlock contention. As will be discussed later, you probably will need to create multiple tempdb's and multiple tempdb groups (15.5) to isolate different application's tempdb requirements from each other.

For example, a select/into can use large I/O's for writing – so it can leverage the 16K pool. The more concurrent users, the more the number of possible processes all attempting to use the 16K pool. This means that most likely tempdb needs a larger 16K pool than 2K if most tempdb actions are select/into or worktable related. Consider the following sp_sysmon snapshot from a 64 engine server:

```
-------------------------------------------------------------------------------
  Cache: tempdb_cache
                                       per sec      per xact       count  % of total
  -------------------------  ------------  ------------  ----------  ----------

…
    Pool Turnover
      2  Kb Pool
           LRU Buffer Grab           45.3           0.3        1450       3.6 %
             Grabbed Dirty            0.0           0.0           0       0.0 %
      4  Kb Pool
           LRU Buffer Grab          622.3           3.6       19914      49.5 %
             Grabbed Dirty            0.0           0.0           0       0.0 %
      16 Kb Pool
           LRU Buffer Grab          589.4           3.4       18862      46.9 %
             Grabbed Dirty            0.0           0.0           0       0.0 %
  -------------------------  ------------  ------------  ----------
    Total Cache Turnover           1257.1           7.2       40226

…
    Large I/O Usage
      Large I/Os Performed         1211.8           6.9       38776      99.6 %

      Large I/Os Denied due to
        Pool < Prefetch Size          0.0           0.0           0       0.0 %
        Pages Requested
        Reside in Another
        Buffer Pool                   4.9           0.0         157       0.4 %
  -------------------------  ------------  ------------  ----------
    Total Large I/O Requests       1216.7           7.0       38933

    Large I/O Detail
     4   Kb Pool
        Pages Cached               1244.6           7.1       39828       n/a
        Pages Used                 1244.6           7.1       39827     100.0 %
     16  Kb Pool
        Pages Cached               4715.5          27.0      150896       n/a
        Pages Used                  613.9           3.5       19645      13.0 %
```

As can be seen by the top three highlighted numbers of LRU buffer grabs, the 16K pool is much more heavily used than the 2K pool – and yet most DBA's configure the 16K pool smaller for tempdb than the 2K pool. The 4K pool usage above is the tempdb transaction log – which is the most heavily used pool – which indicates a large number of logged I/O operations – possibly procs that have a create table #temp(..) followed by an insert/select vs. the standard select/into. With ASE 15, this logging can be eliminated through increasing 'session tempdb log cache size' – in this case minimally 8 pages (16K) would be recommended as the average pages cached per transaction in the 4K pool was 7.1 (rounding up to 8 pages).

While the above illustrates the more heavy use of tempdb's 16K pool, fortunately, this buffer size was set high enough that none of the pages were grabbed dirty. There are several reasons this could happen. One reason would be that a stored proc created a temp table and didn't drop it as soon as finished, but rather waited for the proc to exit – by which time, multiple temp tables resided in the cache. The second reason was the topic for this section – high numbers of cache partitions coupled with smaller pool sizes could result in pages hitting the wash much more frequently than if lower cache partitions or larger pool sizes.

An interesting aspect is the tempdb transaction log. If we set the 'session tempdb log cache size' to 16K or 32K, if the ULC gets flushed to the transaction log due to being full, it will take multiple transfers if the log cache is only using a 4K pool. Ideally, a larger pool size might be more effective. The 4K default log I/O size was chosen to align with the 4K frame size/block size common to most devices. However, a larger buffer size such as 8K or 16K would still be effective and may even be more effective. There are two problems with this:

- Tempdb is rebuilt from model after each reboot – consequently the log I/O size would need to be reset each time the server was rebooted.
- If trying to use the 16K pool, the log pages may push temp tables out of cache

With respect to the latter, it is likely always best to keep the log cache pool size and the large IO pool size separate to avoid the problem.

Restriction Cache (BLOB's, etc.)

One commonly overlooked caching issue is that pages recently written will hang out in cache until they hit the wash marker and are reclaimed at the end of the LRU chain. As a result, some data fields which are often written just once but then rarely read – such as text or image columns – can consume a large amount of the data cache. This has a double impact in that not only does it waste a lot of memory, but it also causes other more frequently hit pages to get pushed from cache. The size of the problem is often surprising – even for DBA's that think such fields are rarely modified. One customer with a 20GB main cache found out that 5GB of it was consumed by image data – some of it from days ago. Another customer who thought that a particular text column was rarely if ever used was surprised to find out that a text column was consuming nearly 1GB of an 8GB cache.

Text and image columns are not the only culprits. Today, with a lot of online reporting, historical data can consume a lot of the cache as well. While caching historical data can improve online reporting speeds, consider the following:

- How does the report's SLA compare to the OLTP SLA?? If the report is allowed to run in a 5 minute window, caching it so that it runs in mere seconds may push the OTLP transactions to the edge of the SLA.
- How often is the same historical data re-read for reporting? For example, if we run a end-of-the week or end-of-month reports once at the end of the week/month, then the data is read and cached for that single report, it will likely either age out – or never get used again until the end-of-month report.

Both text/image as well as historical data can be bound to a "restriction" cache. The size of this cache depends on the caching requirements for the data. For some systems, a restriction cache of 50MB is sufficient. Others may need a 500MB or even 1GB restriction cache. Because a restriction cache will likely have higher turn-over (volatility) in the pages cached, a restriction cache should use the standard 'strict' cache replacement policy vs. a 'relaxed' cache. However, it may need to have a larger 16K pool that one might think. Historical data used for reports is often subject to range scans – either index or data pages – which can benefit from large I/O's. However, remember, that new rows being appended to the historical tables as well as text allocations will likely be out of the 2K (or server pagesize) pool. After creating a restriction cache, careful monitoring will be needed initially and periodically to observe pool usage as well as watching for cache stalls.

An interesting point to consider is whether or not to allow the housekeeper to run in the restriction cache. If a heavy hit OLTP transaction history table is bound to the cache, then the housekeeper might be a good idea. If primarily used to restrict text/image, however, the housekeeper may not be needed. Remember as well, that due to the size of the cache and cache volatility, pages will hit the wash marker faster – therefore the housekeeper may not be as necessary.

Reference Table Cache

Another small cache that is necessary is the "reference" table cache. This is extremely useful for those lookup code tables, as well as other tables frequently referenced but fairly static (e.g. products). The rationale for this is that while they don't take up a lot of memory, they can be frequently flushed from cache – which slows inserts, updates and deletes (due to RI constraints that point to these tables) – as well as inserts.

This cache can be fairly small – a quick check of the space used by the reference tables in the database could give you a pretty good clue of the maximum size you need. Note that for RI constraint coverage, you might not need to cache the table – just the primary key index – especially if the table is DOL locked or the pkey index is nonclustered. In those cases, the pkey index would be all that you need cached. This is true for tables such as products – which all you might need to cache is the pkey index on product_id and the product name index. If someone wants the description or dimensions of the product, the table itself could still be in the default data cache or restriction cache (above) which would allow it to age out when no longer being viewed.

Since the data in this cache is fairly static, the cache should have a relaxed cache strategy. In addition, it likely would not benefit much from a large 8x (e.g. 16K) pool – especially due to the

size. One problem with large buffer pools in small caches is that they end up being fairly small and as a result, you get more turnover for tables that use the pool vs. a single large server page sized pool. Because the pages are fairly static and should have low volatility from a cache turnover perspective, there really isn't much need to partition it very much either.

Hot Table Cache(s)

In addition to the oft overlooked cache consumption of BLOB columns, one of the other most common caching issues is the lack of a dedicated cache for hot objects. Consider the following table pulled from a real customer site during 3 hour sample of monOpenObjectActivity:

TableName	IndexID	Logical Reads	Physical Reads	APFReads	UsedCount
Table_1	0	21,517,184	0	19,245,616	6,968
Table_2	0	8,646,750	0	8,307,689	14,129
Table_3	0	2,058,968	0	2,058,969	1,029,486
Table_4	0	28,258	0	28,258	14,129
Table_5	0	14,312	0	14,312	7,156
Table_6	0	14,312	0	14,312	7,156

Fortunately, the number of PhysicalReads=0 or else the first two tables with the frequent table scans would really drive the IO subsystem hard with a lot of APF prefetch reads. Unfortunately, since these are all in-memory, the result is a much higher CPU utilization than is necessary. Finding and eliminating those table scans would greatly help the system. But there is a second problem indicated in the highlighted tables. These tables are extremely small – Table_3 and Table_4 are only 2 pages, while Table_5 & Table_6 are a single page each. The problem is that being so small, every table access is a table scan. Tables 4-6 are not so much the problem as Table_3, which is causing (literally) millions of LRU-MRU relinkages – and undoubtedly high contention for cache spinlocks.

On closer questioning, it was determined that Tables_5 & 6 were sequential key tables – with a single row. Tables_3 & 4 were similar – although not sequential key tables, they were tables that were constantly scanned due to RI constraints on tables that were heavily accessed. For example, Table_3 was never updated, deleted or inserted into during the 3 hour run. This would be common for a small reference table – for example state/providence codes – that would fit on 2 pages due small row size, and with APL, the clustered index leaf pages would be the data pages, hence just as quick to tablescan. Tables_3 & 4 therefore are best moved to the reference cache using a relaxed cache strategy. Tables_5 & 6 are best placed in a dedicated cache for hot tables.

The above (key sequence tables) is one example. In other cases, you will have extremely volatile tables – such as job queue, call center queues, pending orders or other tables that sustain a lot of insert, update and delete activity during the day. While they can be extremely small – they are highly volatile. A third example is the typical transaction table which sustains heavy inserts appended with probably equally heavy updates (batch or OLTP) and rare deletes (only during archiving). Consider each of these examples:

Key Sequence Tables – Heavy updates with a lot of contention for same rows/pages.

Job/Workflow Queues – High insert/update/delete activity moving linearly through the table. Selects to get next job often lack suitable predicates for indexing, consequently often tablescan to find the next pending job.

Key Transaction Tables – Heavy inserts and possibly updates to recently inserted rows – appending to the table (or deliberately scattered). Often used for selects on DRI on child object tables. Reports may access older data.

We really are discussing two different caching profiles. For key sequence tables, cache partitioning is really not going to help as the users are all after the same few pages. For the job/workflow queue and key transaction tables, partitioning will be key as it help split some of the contention between MRU→LRU relinkages of updates on earlier rows from inserts – however a lot of partitions is not likely to benefit as the appending nature of the transactions means that most inserts are after the same "last" page. Certainly, this would be true of any date index on such tables – which is almost a given. If the inserted rows were scattered (clustered on something different than a increasing transaction id or date), then partitioning would be much more helpful. In both cases (workflow queue and key transaction tables), the data page has a relatively short life cycle – it is inserted and most updates occur within a very short interval. After that point the data is fairly static with the exception of final status updates or similar operations by batch jobs run much later.

The net result is that you will likely need 1 and possibly 2 hot object caches. One with no partitions for key sequence tables and other small highly update centric objects. The other would be for the workflow queues and key transaction tables. In the first case, no large buffer pool would be effective. In the second case, the size of the large buffer pool could have some interesting impacts:

- A small large buffer pool would restrict the cache flushing of current data caused by reports that might need to access older data and does so using a range scan or other large IO mechanism. Note that randomly accessed pages would use the small buffer pool.
- A large buffer pool would force fairly recent pages that have finished their short life cycle to disk and provide more caching for reporting queries.

The fun, of course, is which balance to strike. Most customers will likely benefit from the first as many of the reports use small I/O's.

Permission Cache

Earlier in this section, we discussed the system metadata that is cached as it applies to objects such as tables and indexes. There is an aspect of the metadata cache where the object metadata impacts users – and that intersection obviously concerns the user's permissions on the tables and stored procedures being used. Consider the following fragment of a sp_monitor_config 'all' output from a 50 engine server (some of the output removed for space reasons):

```
Name                        Num_free    Num_active  Pct_act Max_Used    Num_Reuse
 -------------------------- ----------- ----------- ------- ----------- -----------
 additional network memory   120773776    36512624  23.21     40735312           0
…
 heap memory per user             8189           3   0.04            3           0
…
 memory per worker process        1545         503  24.56         1511           0
 number of alarms                    0        7001 100.00         8563           0
…
 number of dtx participant         500           0   0.00            4           0
…
```

```
number of remote connecti          20           0   0.00          12           0
number of remote logins            20           0   0.00          12           0
number of remote sites             10           0   0.00           2           0
…
number of user connection        5107        6893  57.44        7530           0
number of worker processe         172          28  14.00         276           0
partition groups                  980          44   4.30          67           0
permission cache entries            0        6268 100.00        6268   750589170
```

This same was taken at 2:30pm –in the middle of the afternoon – coinciding with a 30 minute sp_sysmon. Before we discuss the permission cache, there are some interesting aspects in the above report.

First, the amount of '*heap memory per user*' is probably set to 8192 – which is double the default of 4096 – and little is being used. This memory likely could be used more effectively for the ULC or session tempdb ULC. Secondly, there are parallel queries occurring ('*number of worker processes*' – 28 active) – which considering this is an OLTP system in mid-day suggests that there is reporting as well as OLTP – a factor that becomes more important in the later section on engine groups and separating workloads. The '*number of alarms*' often coincides with 'waitfor delay' T-SQL statements. Considering the large number of alarms active, it suggests that the application incorporates time-out logic and retries. If necessary, this is fine – but often suggests an extreme point of contention that should be looked at. Finally, the very small number of distributed transaction involvement suggests either an application is using distributed transactions when not necessary or unintended. For example, systems that were developed in ASE 11.x days often used RPC's calls between servers. In ASE 12.5.x and higher, these RPC calls are now transactional. While this does provide the ability to rollback the effects of the RPC, if the application was designed around a non-transactional RPC, there is some additional overhead involved that could be impacting recoverability as well as scalability of both systems. The fact there are remote connections suggests this is a possibility. Frequently, such RPC calls are embedded within another stored procedure, which can simply be modified with 'set transactional_rpc off' to avoid this.

Finally, the 'permission cache entries'. The very high amount of reuse and the fact that all are 100% active suggests that when user queries are being executed, the chances are extremely high that the process first has to read sysprotects to determine if the user has permissions. This could have significant impact if the number of users and objects are fairly high as the object permissions would first have to check the individual's permissions, the permissions associated with any roles or groups they are a member of, and then permissions allowed to the public users. With the current security requirements necessitating discrete user logins, it is highly possible that this sysprotects lookup also might require a physical read from disk due to system tables being flushed from cache (see discussion on named caches later in this document). Compare the above with the results from a different customer:

```
Name                       Num_free    Num_active  Pct_act Max_Used    Reuse_cnt
-------------------------- ----------- ----------- ------- ----------- -----------
…
permission cache entries          906          94   9.40           94       90109
```

Note that while the max used is still effectively 100% of the active and the reuse count seems high, the percent active is less than 10%. This suggests there are a few users or transactions that impact a wide range of tables or procedures, but overall minimal impact on processing overhead.

In ASE 12.5, the default 'permission cache entries' was 5. In ASE 15.x, this has been raised to 15 – which is likely barely sufficient for a single transaction in a large production system. For example, a single FSI derivative trade may easily involve 30 or more tables.

The best recommendation is to simply set the configuration value to 25 and monitor with sp_monitorconfig focusing on percent active – with a goal of keeping this below 20-30%. Additionally, monitor sysprotects in monCachedObject and observe any fluctuation in the amount cached (sysprotects cannot be bound to a named cache, so this will be in default data cache). If it appears that physical reads are happening due to frequent fluctuation in the amount of sysprotects cached, you may also want to increase the configuration value. Typically, any increase should be in increments of 5 or 10 at a time.

Configuration Parameter	Default	Recommendation	Tuning Considerations
permission cache entries	15	25	Increase in increments of 5 or 10 if sp_monitorconfig shows significant percent used (20-30%).

Application Specific

Application specific caches are often necessary as a result of server consolidation. While we would like to assume that two applications can peacefully coexist – the reality is that errant queries or adhoc analysis can lead to table scans – which not only affects the data cache for the application that is doing the querying – it affects the data cache for all other applications as well. Even without errant queries, simple DBA maintenance such as table reorg, update statistics or dbcc checks can cause the cache to be flushed as it performs a table scan.

Application specific caches are often used in lieu of the default data cache. In other words, entire databases that comprise the application are often bound to a separate cache. In some cases, DBA's create multiple application specific caches and have a much smaller default data cache as a result.

Another form of application specific cache is when a cache is created to handle common data for different application modules or key performance data for mission critical application modules. For example, for flexibility reasons, often application security permissions are stored in the database. This application permissions are often used to determine whether buttons are 'grayed out' or not, as well as often used to provide the where clause to filter data appropriate for the user. Because of the frequency of use, application security data often is bound to a separate named cache.

Whether or not the cache is partitioned or the cache strategy is strict vs. relaxed often depends on the data. For example, in the former case of separating two competing OLTP applications, the application specific cache may look similar to the default data cache – heavily partitioned, strict cache replacement policy. For application caches similar to the permissions example, it may look more like a reference data cache with fewer partitions and using a 'relaxed' cache replacement policy.

Named Cache Summary

The following chart summarizes the different recommended cache configurations discussed in the above paragraphs.

Named Cache	Sizing	Number	Partition	Cache Strategies	HK Ignore
Log cache	50-100MB (normal) 150-200MB (XXL SMP)	1-3	No	Log only, Relaxed	(implicit)
System tables	200MB-500MB	1	No or few	Relaxed	(implicit)
Tempdb caches	250MB-500MB (normal) 500MB-1GB (XXL SMP)	1 per tempdb or tempdb group	YES*	HK Ignore Cache	YES
Restriction	50-100MB (normal) 256-500MB (BLOB)	1	YES*	Strict (default)	(maybe)
Reference	50-100MB	1	No or few	Relaxed	(implicit)
Hot Tables (static size & fixed length fields - update intensive) such as key sequence tables	10-50 MB	1	Few or more*	Relaxed	(implicit)
Hot Tables/Indexes	Size of volatile data	As necessary	Few or more*	Strict (default)	NO
Application Specific	As necessary	1-3	YES*	(depends)	NO
Default Data Cache	(most of memory)	(1)	YES*	(default)	NO

YES - Partitioning should be based on spinlock observed contention. If none, consider decreasing local partition number until slight contention appears and then adjust to next higher value.*

As with any starting recommendation, optimal configuration will require monitoring and tuning the system to the application requirements.

Multiple Tempdbs & Tempdb Groups

Tempdb is one of the more heavily used databases within the Sybase ASE instance. It is used for worktables for joins, sorts, reformatting and other query processing requirements, application temporary tables (i.e. #temp), and system maintenance tasks such as update statistics. While tempdb is generally non-recoverable, it still supports transactional DML statements. Therefore, a transaction log is still maintained - although by default it is truncated each checkpoint interval. The phrase “generally non-recoverable” was not deliberately intended to confuse over boot time trace flags that don’t clear tempdb – but rather consider that in ASE Cluster Edition, global tempdb shared tables are recoverable in the event of a node failure – allowing processes using shared temp tables to failover without loss of any temporary data.

Ignoring that aspect and focusing on the traditional ASE tempdb, the biggest problems with tempdb that DBA's typically have to deal with include:

- *Speed*. Since these are often intra-query work tables or transient application temporary tables, they do not need to be persisted on disk. Ideally, all of tempdb would fit in memory, however, this impractical and DBA's need to be conscious of letting large temp tables "spill" to disk
- *Size*. Tempdb needs to be appropriately sized to avoid query failure or log suspension. This can be caused by improper query formation resulting in full or partial Cartesians that fill space - or simply due to an unexpectedly large result set (or higher concurrency).
- *System Catalog Contention*. Up through at least 15.0.2, concurrent requests to create temporary tables results in contention on the system catalogs in tempdb
- *Log Contention* - While many strive to use minimally logged operations, the logging of allocation pages, normal ULC flushing and fully logged DML operations can lead to contention on the single log semaphore.

The best way to setup tempdb often becomes quickly controversial as any number of DBA's will have their own ideas based on their experiences, the application profile, gouge they have heard from others - or simply working within the restrictions of the corporate standards adopted by the organization they work for.

The reality is that the best way to set up tempdb will vary from system to system depending on the applications involved. However, there are common patterns. Before we get into those patterns, we need to discuss some recent changes in tempdb, operating systems and hardware as many long time tempdb implementations are actually causing problems now – especially those built on shared memory file systems.

Recent Enhancements in Tempdb Optimization

One of the biggest drivers in how tempdb was implemented in the past often was dictated on how tempdb behaved – in particular, the system catalog contention. Over the past ~6 years, Sybase has added a number of enhancements that strive to improve the performance of tempdb and remove it as a restriction on overall system throughput. Note that the

optimizations discussed below apply to both user-created and system temporary databases (see discussion on multiple tempdb's later).

Row Level Catalog Locking

Arguably, the biggest enhancement for tempdb came in ASE 15.0.2 when ASE 15's row level locking for system tables was finally enabled. It had been implemented prior to that point, but not enabled to ensure a thorough cleaning of all the code for hardcoded direct accesses to system table pages had been removed and replaced with the logical references to the table names. Prior to ASE 15.0.2, tempdb contention came under a lot of fire – justifiably in most cases – due to contention between concurrent users. While multiple tempdb's alleviated the contention in many cases, in other cases, the lack of array datatypes and in-memory database capability forced application developers to use temporary tables for high volume processing that drove contention extremely high.

However, in just as many cases, tempdb contention took the blunt of the blame, when application design was even more a culprit – orders of magnitude more the culprit. For example, in one case a customer was attempting to scale from 50+ engines to 60+ engines. The increase was due to a change where formally micro-batch processed transactions processed every few minutes would be processed in OLTP form as they occurred. The goal was to reduce the business latency and customer wait times for transactions. The result was a severe performance degradation – blamed largely on tempdb system table contention. This was demonstrated by an in-house developed utility procedure that would show the number of users waiting on locks (blocked) by table – run by the DBA's manually during the problem. However, a closer look by constantly polling (no delay) monLocks showed the following lock waits as the top 3 causes of contention (comparing normal throughput vs. degraded):

Table	Normal	Degraded
Main transaction table	8,289	3,265
Key sequence table (key=key+1)	46,070	336,769
Tempdb system table	1,238	10,464

Yes, there was a 10x increase in tempdb system table contention. However, it still was 30-40x less than the contention on the key sequence table. Part of the suspected problem was that in the transition from batch to OLTP – whereas before the batch process could preallocate 10's-100's of key sequences needed, now each of the individual transactions was attempting to allocate a single key. Even in the degraded scenario, the tempdb system table contention was 4x less than the contention on the key sequence table under normal circumstances. Unfortunately, the key sequence grabbing logic was embedded in SQL within the application code vs. a store procedure interface. As a result, common quick fixes to the key sequence table problem could not be used until a complete application rewrite. The overall result: because of a poor application design, a business goal could not be achieved.

However, in just as many cases – especially where multiple tempdbs were not used – tempdb contention indeed was the main culprit. Consequently, the RLC implementation and its impact on tempdb will likely benefit more applications than the rest of the tempdb optimizations combined for sites with only a single or few tempdbs.

Select/Into Optimizations

Beginning with ASE 12.5.1, Sybase began adding special optimizations within tempdb for select/into - a common operation for creating and populating application temporary tables. These enhancements include:

- Use of a large buffer pool available for the target table
- Use of extent allocation instead of page allocation
- No logging for in-memory catalog updates for the last page
- Avoiding data writes and transaction log writes to the disk at commit time

The last bullet may cause some confusion as many think that select/into is not logged. In reality select/into and other bulk operations are considered minimally logged - which means schema modifications, space allocations and other non-data changes are logged to ensure database recovery (vs. corruption). In a normal recoverable database, as data changes are written to the transaction log, the log writes are flushed to disk immediately vs. awaiting a wash marker, checkpoint or other induced flush. The reason is that Sybase, like most DBMS vendors, uses a concept of *write-ahead logging* - in which data modifications are first written to the transaction log *prior* to the transaction being allowed to commit. Even though select/into queries are typically not within an explicit transaction, each select/into represents an atomic transaction - consequently each select/into often times involved at least one if not multiple log writes - each of which would be dependent on the speed of the transaction log device for query response times.

Lazy I/O for Log & Data

Again beginning with ASE 12.5.1 and throughout 12.5.2 & 12.5.3 releases, Sybase started processing tempdb more like an in-memory database. As noted above, this began in 12.5.1 for select/into queries, which would attempt to bypass all physical writes if possible. More than just select/into, however, starting with ASE 12.5.1, the server began minimizing the I/O by not issuing IO's on log or data page writes during the normal transactional code path involving creation of work tables and DMLs on temp tables. Additionally, if a temp table was dropped prior to the cache wash marker or checkpoint process where an IO would be forced, the physical writes would be avoided completely. In ASE 15.0.2, this was enhanced such that the checkpoint process would no long flush dirty data pages to disk leaving the wash marker as the sole remaining point at which physical IO's would normally be scheduled for dirty data pages. Note that this cannot be eliminated entirely as large temp tables or high concurrency could cause the data cache to fill necessitating removing pages from cache. This spill over needs to be monitored and prevented to the extent possible (see discussion on tempdb named caches).

Session Tempdb ULC

Normally a user has a single Private Log Cache (aka User Log Cache) which caches log records until the transaction commits, the cache was exceeded or any number of system events. Some of those system events include:

- *Transactions involving both a user database and tempdb*. ASE flushed the user log cache (ULC) as user sessions switched between transactions in the user databases and tempdb database. For example, if the user modified a user database table, then did a select/into tempdb, and then updated another user database table based on the tempdb that was created, there would be minimally 3 ULC flushes - one between the first modification and the select/into, one after the select/into and one after the second user database modification. These flushes happened irrespective of the number of rows modified.
- *Buffer unpinning* – As discussed earlier, if a user modifies a database page, that page's buffer is "pinned" to the ULC. If the user's transaction rollsback, it is faster to find the page and undo the modification as a log scan is not required if the ULC has not been flushed. Remember, the buffer (data page) is "dirty" - and most likely it has not been written to disk. If another user modifies the same page, it has to be "unpinned" from the first user. The assumption is first user's transaction has not yet committed and that the writes are non-conflicting (i.e. datarows locking). Unpinning causes the ULC to be flushed. In addition, a transaction rollback now takes longer as a log scan is required
- *Single Log Records* (SLR) – As also discussed earlier, ASE creates an SLR for any allocation page (OAM) modification in databases that have mixed data & log allocations (i.e. tempdb). SLR's force a flush of the ULC immediately to avoid potential internal deadlocks during "unpinning"

In ASE 15.0.2, a number of enhancements were added to the server to avoid these potential restrictions on tempdb throughput:

- A session tempdb ULC was implemented. ASE 15.0.2 includes a separate ULC for the session's temporary database, so multi-database transactions that include a single user database and the session's temporary database do not require ULC flushes when the users switch between the databases.
- Buffers are no longer pinned in non-recoverable databases such as tempdb. This allows the session tempdb ULC to be used more effectively and possibly prevent a lot of log flushes. This may increase rollback time on shared temp tables.
- ASE 15.0.2 no longer does a ULC flush for SLR's, buffer unpin, commit transaction. Some of this was simply avoided by not pinning buffers in non-recoverable databases such as tempdb.

With multiple tempdb's possible, this can get quite confusing – especially in ASE/CE. Remember, it is the user's session tempdb – the one where #temp and worktables are created – the utilizes the session tempdb ULC. It is possible that a user transaction involves a shared temp table from a different tempdb or a global tempdb (ASE/CE) and as a result, they still would use the normal user's ULC.

Page Splits & Tempdb

A page split is caused by a row expansion or an insert according to a placement (clustered) index. In each case - either the row expansion or the newly inserted row, causes the amount of data on the page to exceed the page capacity. When Adaptive Server splits an index or data

page, it moves some rows from the original (modified) pages to the newly created page. While the data modification that triggered the page split is logged, the rows that were moved to the new page are not logged. To ensure consistency of the database and ability to rollback a transaction, ASE needs to ensure that either the write of the new page to disk occurred ahead of the modified page or that the new page reaches the disk before that transaction commits. Prior to ASE 15.0.2, this was implemented by using synchronous IO's during page splits. Note that the synchronous IO's were from the perspective of the user's session. Internally, ASE would still use the OS's asynchronous IO libraries to submit the write to disk. However, the user's session would block until that IO returned – much like the blocking commit for the last log page. As a consequence, ignoring log writes due to ULC flushing, a normal process modifying pages would quickly move through the pages being modified – updating the cached versions with no physical IO waits. A process that involved a lot of page splits, on the other hand, would essentially "stall" every time a page split occurred and wait for a physical disk IO. The disk write speed was only part of the issue. Remember, a process is only put to sleep in ASE when a physical IO, network IO or blocked lock was required. So, in the first case, the process could use more of its CPU timeslice and finish quickly. In the case of page splits, the process frequently went through a cycle of execute → page split → sleep → wait for engine → execute.

In ASE 15.0.2, ASE uses an asynchronous method of ensuring database consistency. While this affects all ASE tables, it can help tempdb performance due to a common application technique. Often times, an application developer will create a temp table with a number of nullable columns. These columns are "filled out" by calling subsequent update statements to populate the values. Unfortunately, by populating these values, the tempdb table's rows expand – most often, every row was expanding. As a consequence, a large number of page splits could occur in temp tables and processing took longer.

Tempdb Storage – SSD, IMDB & File Systems

There are multiple aspects to tempdb physical storage – and as many myths, half truths and a lot of unsubstantiated web gouge. The principal aspects of tempdb storage can be divided into the following categories:

- File system vs. Raw devices
- Physical Storage: SAN, SSD or DASD
- Tempdb creation options

The following sections detail each of these aspects and wraps up with a recommendation.

File System vs. Raw Devices

> *NOTE: In this discussion, the term UFS is meant to refer to any unix file system generically - whether UFS, JFS, ZFS, ext2, ext3, reiser, xfs, etc. unless specifically cited (e.g. ZFS).*

The simple fact is that raw devices provide the highest performance for *write* activity, ensure data integrity and are the easiest for providing consistent performance. On the other hand, they are not as fast for *reading* – especially *sequential reads* - as file system devices – which

have highly optimized read-ahead logic in the file system driver. Additionally, once sized, it is hard to shrink or grow a raw device as it involves modifying the physical layout of the physical disks or LUNs. The first point is the most applicable for tempdb as tempdb incurs a lot of table scans as few #temp tables have indexes. In addition, most OS's set aside considerable amounts of main memory for file system caching – which can help speed reads and writes if the data is not recoverable and the cache can be exploited. Using file-based cached devices can provide considerable performance gains - however, there is considerable setup that must be done to ensure the performance gains can be realized.

A crucial step that is often overlooked is that a cached file system device is a Sybase device that is created on a file system with both DIRECTIO and DSYNC options off. This is not the default as typically for file system devices, DSYNC is enabled - which seriously degrades performance. This setting can be checked or set for each device either via the disk init command when the device is created or using the sp_deviceattr stored procedure after the device has been created.

Beyond the device DIRECTIO and DSYNC attribute there are a number of important factors about setting up tempdb on file systems. Failure to consider these aspects could lead to serious performance degradations. While Sybase has stated in various technotes, etc. that "Using file systems for tempdb *may* [italics and underlining the authors] improve performance", what was not stated was that this performance gain was highly dependent upon optimal configuration of the file system and in some cases was predicated on aspects that no longer apply in ASE 15.

File System Support for Asynchronous IO

Not all file systems support asynchronous IO. For example, the native HP filesystem in HP-UX does not support asynchronous IO, but the bundled Veritas™ file system used to support journaled file systems under HP-UX does support asynchronous I/O. This can be crucial in higher concurrency environments. Within the ASE process, a number of threads are spawned at the OS level to handle network and disk IO requests. When a physical IO is necessary, one of these threads simply makes the appropriate OS library calls to submit the IO to the operating system. If the file system does not support asynchronous IO, the result is a synchronous/blocking IO call in which the calling thread is put to sleep until the IO returns. While the ASE engines continue to run, the number of threads available for IO processing have effectively been reduced. The more blocking IO calls that are pending, the system throughput starts to degrade for tasks awaiting physical IO's.

One large trading company found this out the hard way. Due to the above cited "Sybase recommendation", they had moved their tempdb to a file system. Some initial testing by the DBA's indicated that it was faster. However, in production, with 900 concurrent users on a 20 engine HP SuperDome, the file system devices were reporting an average 80ms service time – 10x what is often considered the upper limit of device service times (8ms). The problem was that at the full production concurrency, the file system cache quickly was flooded and the net result was that the next file system IO could not be handled until an earlier one was destaged. This not only lead to slower processing, but also resulted in much higher tempdb contention. Changing back to a raw partition nearly eliminated tempdb performance issues. A similar option would have been to switch to a file system that supported asynchronous IO.

Another aspect is that a file system may be asynchronous up to the point of the request queue size. Once that request queue is full, it may block other requests – which will either result in

the requesting process being put to sleep or go into a retry spin loop and consume CPU. For example, on some Sybase engineering internal tests with Linux, it looked as if the ASE engines were pegged at 100% from sp_sysmon, while vmstat showed little activity. The problem was traced to the fact that on Linux, the default /sys/block/*/queue/nr_requests is set to 128 (Compare this to Solaris which supports 2000 concurrent IO requests per volume by default). Under heavy IO load, the OS would block in io_submit() calls as the block device would wait until the number of outstanding IO's dropped below the threshold. The OS would put the process to sleep (hence the low utilization in vmstat), but the ASE engine (when awake) would be constantly trying to schedule the IO.

The point of these two examples is that you really need to test the filesystem under *full* production *peak* load to understand what tuning may be necessary to avoid performance degradation as the user load scales.

Using Shared Memory File Systems

Another common file system that does not support asynchronous IO is most shared memory file system devices - such as tmpfs (Solaris) or shmfs (Linux). While shared memory IO should be extremely fast, the OS kernel tuning for IO polling times (see the section on OS kernel tuning) may restrict the responsiveness leading to the same situation described above. Currently none of the major platform OS's support an asynchronous driver for their shared memory file system, although one reportedly is under development for the Linux kernel.

Notes on tmpfs

Unfortunately, many DBA's simply assume that putting tempdb in tmpfs eliminates all their problems. It can – if the overall system performance is taken into consideration up front and both the DBA's and system administrators monitor swap activity on the system to prevent any unforeseen problems. Consider the following bug report:

> *Tmpfs is a memory resident file system. It uses the page cache for caching file data. Files created in tmpfs file system avoids physical disk read and write. Thus the primary goal of designing tmpfs was to improve read/write performance of short lived files without invoking network and disk I/O.*
>
> *Tmpfs does not use a dedicated memory such as "RAM DISK", instead it uses virtual memory maintained by the kernel. This allows it to use VM and kernel resource allocation policies. Tmpfs files are written and read directly from the kernel memory. Pages allocated to tmpfs files are treated the same way as any other physical memory pages. It means tmpfs pages can be freed and put it back to the backing store during memory shortage.*
>
> *It is the system administrator responsibility to keep a back up of tmpfs files by copying tmpfs files to disk based file system such as ufs. Otherwise, tmpfs files will be lost in case of a crash or a reboot.*
>
> *When tmpfs file is created or modified pages are marked dirty. Tmpfs pages stays dirty until the file is deleted. Only time tmpfs_putpage() routine pushes the dirty tmpfs pages to the swap device when the system experiencing the memory pressure. System with no physical swap device or configured with plenty of physical memory can avoid this overhead by setting tmpfs tunable "tmp_nopage = 1" in /etc/system file. Setting this tunable causes tmpfs_putpage() to return immediately without incurring the overhead of searching the dirty page in the global page hash, locking the page, finding out the backing store of the page and unlocking the page in case no physical swap is configured and system has plenty of memory.*
>
> *http://bugs.opensolaris.org/bugdatabase/view_bug.do;?bug_id=5070942*

In addition to this, over the years, a number of sites have reported ASE crashes when using tmpfs for tempdb due to the OS deleting the tempdb devices. The reason this happens is that when the OS is under pressure for swap space, in some circumstances, it simply looks for and removes the oldest files in swap. The authors suspicion is that this was more likely to happen if

the device was simply created in /tmp vs. creating a dedicated tmpfs device via a mount point in /etc/fstab. There is a distinction in that creating a device in /tmp really is not using tmpfs, but rather swap. See discussion later on in-memory tempdb for the details on setting up tmpfs on Solaris.

In Linux, one aspect to consider as well is the swappiness kernel parameter (vm.swappiness). The default is 60 and indicates a preference for unmapping (or swapping pages) vs. a lower value. If using a tmpfs device and you start to notice an increase in swapped pages, it may not be due to an over allocation of memory, but simply this parameter being set too high. Reducing it may help. To monitor swap activity, use the vmstat command and focus on the 'si' and 'so' columns.

tmpfs vs. ramdisks

An interesting and little known aspect is that in addition to supporting tmpfs, Solaris also supported a true RAM disk file system using ramfs that could be created using the ramdiskadm similar to the following:

```
# list all the ramdisks on a system
/usr/sbin/ramdiskadm

# create a ramdisk for tempdb 10GB in size
/usr/sbin/ramdiskadm –a SRV1_tempdb 10g

# fix owner/group so ASE can write to it
chown sybase /dev/ramdisk/SRV1_tempdb
chgrp sybase /dev/ramdisk/SRV1_tempdb
```

The above creates a file in /dev/ramdisk named SRV1_tempdb. The key advantage to ramdisk over tmpfs is that it is truly memory vs. created out of swap and as a consequence are never swapped out due to virtual memory requirements. An advantage of tmpfs is that it is common to a lot of Unix based operating systems, including Linux, whereas ramdisks differ and that it is limited only by the amount of *virtual* memory (swap space). For example, older versions of Linux supported the notion of a ramdisk, but as a block device (void of file system) which means you have to format it with mkfs (using ext2 as journaling an inmemory file system is useless). Additionally, Linux creates 16 small ramdisk on boot – none of which have any memory allocated. To increase the size, you would need to change grub.conf. Later bug reports on Linux offer confusing comments whether this is still supported, renamed to tmpfs_size or simply deprecated. However, the devices are still there in RHEL 5.x as is visible with a simple ls –l /dev/ram*, and the general recommendation is to use tmpfs instead.

Current Applicability of tmpfs

One aspect to consider is that as Sybase ASE has been dramatically reducing the amount of physical IO's used in tempdb, some of the critical drivers for certain implementations in the past are no longer a consideration. In addition, even during moderate testing, problems may not be visible – it may require much high concurrency simulations to drive the same test accuracy (e.g. high hundreds of concurrent users). For example, one of the common tempdb testing scenarios in the past was to simply have a large number of concurrent users create temp tables – either empty or by selecting from a system table. For example, a common tempdb scalability test in the past might resemble:

```
-- execute by a large number of concurrent threads
create proc tempdb_test
as begin
        select * into #mytemp from sysobjects where 1=2
        drop table #mytemp
end
go
```

In the past, this used to reflect the system table contention issues fairly easily. However, with ASE 15 not logging allocation pages for non-recoverable databases – and with the session tempdb log cache all but eliminating writes entirely on small tables or fairly decently sized minimally logged tables (e.g. select/into) – the amount of physical writes to the underlying devices has reduced dramatically. Along with it has been the reduction to use special devices such as shared memory file systems or even SSD's (more on this later).

Initially, one of the driving reasons for using a shared memory file system was due to the system catalog contention in tempdb. As a result, the contention was often reduced as the system catalog writes were much quicker and the locks released faster. However, with ASE 15.0.2's RLC implementation, this reason was eliminated. In fact, after upgrading to ASE 15.0.2, many customers using shared memory file systems started noticing a lot of mutex contention within the file system (on Solaris, this contention can be spotted with the dtrace command). The reason was that on Solaris, asynchronous IO calls to file system devices are implement as POSIX threads using Light Weight Processes (lwp) instead of kernel threads. The threading difference was not the issue so much as these concurrent lwp processes were all trying to do concurrent IO to the same (shared memory) synchronous file system – and were getting blocked. In most cases, simply switching to a normal cached file system alleviates the problem. In other cases, switching to an in-memory database for tempdb is the better solution.

This brings up the second most common driving factor for using a shared memory file system - DBA's could try to simulate an in-memory database option for tempdb. Some today, still feel that this is a viable alternative to the in-memory option provided by ASE. There are multiple problems with this assumption:

- First, of course, there is the issue of a synchronous file system
- You need up to twice the amount of memory to avoid physical IO's as much as possible from ASE's perspective

The second point may sound a bit confusing, but remember, whether you have a dedicated cache for tempdb or not, ASE will cache tempdb tables in its memory. Of course, if you create a dedicated cache for tempdb, all the better, as mentioned earlier – as it can be optimized (no HK, etc.). Regardless, what you actually have is not an in-memory tempdb, but a *memory-mapped* device for tempdb. This can lead to some interesting trade-offs. For instance to have a 10GB tempdb, you would need 10GB in shared memory file system, plus a considerable amount of memory in ASE cache to avoid ASE thinking it needs to do a physical IO. The problem is that if using a shared memory file system – even for a part of tempdb, ASE isn't aware of the fact it is memory – it thinks it is a disk device. Consequently, when a physical write happens, the process is still put to sleep in ASE. ASE will also poll the IO on the same polling interval as other devices – and the OS kernel settings often will dictate the response times on the polling as well.

Keeping this in mind, a shared memory file system may have significant advantages during low tempdb concurrency - however, the memory may be much better utilized to support overall file system caching or a database named cache dedicated to tempdb. Interestingly, since a shared

memory file system does not support asynchronous IO, it becomes a good candidate for tempdb devices that are serialized - currently the transaction log. However, it is doubtful that large shared memory file systems have any substantial benefit over allocating the same amount of memory for file system caching and using cached file system devices for these same Sybase virtual devices.

File System Cache Size

If the goal is to reduce the likelihood of physical IO's, the amount of memory available for file system caching is *critical* when using UFS devices. Trying to use cached UFS devices with too little memory can result in swapping out the ASE instance to provide the memory space for the file system cache. Obviously, this could have a serious impact on performance.

Normally, if the host is dedicated to the DBMS server, the amount of memory configured for the DBMS instance will often be roughly 70% of the total memory in the machine. This leaves about 30% for the OS and the file system caching - which may be barely enough for the OS. Consider a typical dual processor machine with 8GB of memory. If the DBMS allocates 6GB of memory, this leaves only 2GB for the OS and file system caching. On high volume environments, the OS will need more memory to track async IO requests and other kernel structures - more than making up for the memory saved by eliminating unnecessary services such as print daemons, etc. As a result, it is likely that a typical OS should have 1-2GB of memory. This leaves no memory on the host for creating cached UFS devices.

On the other hand, consider the typical production system for larger SMP machines. Typically, these will have 48 to 64GB of main memory. With ASE using 32 to 48GB of memory, the other 16GB is available for the OS and file system cache - which is sufficient for 14GB of file system cache - and tempdb device caching. File system caching (or as it is sometimes called Unix Buffer Cache) benefits disk reads and writes in several ways. First, when reading a file, the OS will automatically start pre-fetching subsequent disk blocks. When writes occur, the writes are cached and then destaged to disk according to write gathering algorithms, cache size and other factors. On the plus-side, since it is a cached write, the application that submitted the write assumes that the write has succeeded and does not wait for the write to physically get to disk.

Most OS's have parameters to control how much main memory is set aside for file system caching. For example, with Win32, it is everything above 2GB of main memory. For other OS's, it is controlled by kernel parameters typically contained in /etc/system or similar configuration file. Note that the typically are multiple parameters that may control not only the file system cache size, but which aspects of the file system cache (inodes, page cache and buffer caches) the parameter controls. A quick glance at some common operating systems and file system cache parameters is included in the table below:

OS	Kernel Parameters
AIX	minperm, maxperm, strict_maxperm (see earlier referenced *Optimizing Sybase ASE for IBM AIX* Whitepaper from IBM & Sybase)
Linux (RHEL 5)	vm.min_free_kbytes

Solaris	bufhwm, bufhwm_pct, ufs_WRITES, ufs_LW, ufs_HW (note that parameters may be different for ZFS)

File system cache utilization can be monitored using the Unix utilities sar -b, iostat, vmstat or other OS specific command. Please reference you appropriate vendor's documentation for setting and monitoring the above kernel parameters. However, keep in mind the following DBMS specific points:

- **Do not allow the file system cache + DBMS memory +4GB to exceed total system memory.** The 4GB is an assumption of other non-DBMS process space requirements (such as OS processes) - the actual number may be higher for larger processes - or if a number of medium processes such as JVM's are executing. This latter could be a factor in Sybase environments as some replication agents, UAF agents and other processes are hosted within a JVM.
- **If no UFS devices are used or if only used with DIRECTIO, limit UFS file system caching.** The rationale for this is that opening large files such as ASE errorlogs (including by monitoring/errorlog scanning processes) and normal file accessing will leverage the available UFS file system cache. This may not always be possible – a lot of OS tasks use file systems. For example, limiting file system cache on AIX 5 often lead to kernel panics if too low.
- **File system cache size does not have to be same size as tempdb UFS device size(s).** Remember, the OS will destage IO's to disk from the UFS file, consequently the cache will be periodically paging data to disk and freeing memory. Having the file system cache size at least the same size as the tempdb device sizes fairly much guarantees that the writes will be all cached. It is also likely that a file system cache of 25-30% of the tempdb size may eliminate 60% or more of the physical writes - simply due to the reuse of page allocations/disk blocks within Sybase ASE.
- **If file system cache is limited, tempdb transaction log devices first.** Note that this assumes a separation of log and data devices. The rationale is that these devices currently have serialized access either due to the log semaphore and therefore the faster the physical IO's return from the Sybase perspective, the quicker the blocking condition can be removed.

File System Type, Mount Options & CPU allocation

In addition to the file system cache, the file system type and mount options need to be optimized for tempdb. Most modern day Unix operating systems use a journaling file system by default. Not only is this a bit of overkill since Sybase ASE does its own journaling via the transaction log, but in this case we are discussing tempdb - which is not only non-recoverable - but also is rebuilt from scratch with each reboot of the ASE server instance. How this is controlled may be due to the type of file system chosen or the file system mount option. For example, IBM AIX offers JFS (32 bit) and JFS2 (64 bit) - both journaled file systems, Linux offers ext2 (not journaled) and ext3 (journaled by default, but tunable via the mount option

'cache=writeback') and Solaris offers UFS (not journaled) and ZFS (not journaled - copy on write protected). Journaling offers some interesting aspects:

- The extra overhead of journaling does not always translate into a performance degradation. In fact, with Solaris using --forcedirectio, it was determined that a journaled filesystem was faster than one without.
- The impact of journaling can largely be reduced or eliminated by ensuring that the filesystem journal is on a different physical device.
- The impact of journaling can also be reduced based on the extent of journaling enabled (such as controllable via the Linux mount options cache=writeback).

One of the other mount options to watch is options such as modification time tracking (i.e. noatime and nodiratime). By default, when using files, any modification to the file updates the file modification date and time in the directory structure. This can cause considerable overhead as the amount of updates to any database device - and especially tempdb devices - could be within milliseconds and register in the millions per hour.

In addition, the filesystem blocksize plays a significant role. In most OS filesystems, the default block size is 4KB. However, most ASE's – especially those that have been around for a while – use a 2KB pagesize. The resulting blocksize mismatch causes the OS to use multiple system calls for each write operation. Tests on AIX, for example, found that it doubled the number of system calls. Other operating systems have experienced similar impact. The result is that filesystems intended to be used for ASE need to be created with the blocksize matching the ASE pagesize and then mounted using the blocksize option.

Finally, keep in mind that with raw IO, ASE uses AIO kernel threads to perform the actual disk writes - so the CPU overhead within the OS is minimized. With file system IO, the OS needs to scan the file system cache for blocks to write as well as submitting the write to disk. Consequently, you *may* need to leave more CPU resources available to the operating system. This can be done by ensuring that you follow the n-1 rule of engines to CPU's/cores or that you reduce the 'runnable process search count' to allow ASE to yield the CPU more quickly to OS or other tasks.

Physical Storage: SAN, DASD & SSD's

The second decision point is where to physically put tempdb. Most DBA's simply put tempdb on the SAN – and these days with cost cutting changes in SAN setups, that is becoming more and more of an issue. The second question is whether to use an SSD for the tempdb and issues that may arise from using one.

SAN vs. Direct Attached Storage Device (DASD)

An extremely important aspect of tempdb devices is the physical location of those devices. Many DBA's and system administrators simply co-locate the tempdb devices with all the other database devices on the SAN. This has several large performance implications:

- *SAN Disk Block Replication*. If the SAN physical disks containing the tempdb devices are disk block replicated, any physical writes to those devices may be

slowed tremendously - especially if synchronous disk block replication is involved.

- *SAN RAID Group Contention*. If the SAN physical disks containing the tempdb devices are in the same RAID group as other data devices, physical reads and writes to all devices in that RAID group could be degraded.
- *SAN RAID Level*. In today's cost conscious environments, most shops are switching to RAID 5 or RAID 6 for the standard RAID level. Both of these have serious performance impacts on high IO load devices with frequent physical reads.
- *HBA Adapter Limitations*. Normally HBA Adapters have a limit as to the number of IO operations per second they can handle or the number of concurrent IO operations supported. For example, common IBM pSeries HBA adapters are often only configured for 255 concurrent IO operations. As most Unix kernels support at least 1024 concurrent IO operations and even logical volumes will support 2,000 concurrent IO operations, it is easy to see how that HBA configurations/limitations could become a bottleneck. However, the definition of concurrent IO is a bit different as from the OS (and ASE) standpoint, it is the number of concurrent outstanding IO operations. How the HBA adapter defines concurrent may differ as it may refer to the simultaneous block transfers. Keeping this difference in mind, the default low configuration may have an impact due to frequently low numbers of HBA adapters per system.

Probably the first three are the biggest issues. On the other hand, care must be taken to ensure that the DASD has sufficient IO capacity and caching (via the file system) to achieve the necessary read/write throughput. While a single SCSI disk may work provided that the file system cache is large enough to prevent writes from occurring, you may need to use several disk striped via a RAID controller with sufficient cache to achieve the desired throughput.

The second and most common problem cited against DASD is the lack of redundancy. This is a real concern, but takes a bit of exploring. First, we are not discussing a logical corruption of tempdb – that would impact ASE no matter whether SAN or DASD. We are essentially concerned with the failure of the DASD device or RAID controllers. To see how much of an impact a HW failure could cause, consider the following questions:

- What is the OS's response if a board fails –does it panic?? (Think filesystem)
- Does the OS, LVM or storage array support mirroring between different arrays??

Remember, DASD does not necessarily mean an internal disk. For example, external storage arrays such as Sun's F5100 flash array are HBA connected – similar to a SAN. This allows host mirroring across HBA and storage domains to support full reliability.

SSD's for Tempdb

If you can at all afford it – and today, it is dirt cheap – the best option for tempdb is to host it on a SSD device or storage array – especially for tempdb's used in batch processing or those hard hit enough that physical reads are required due to cache turnover. The common reason that

DBA's may be reluctant to do this was mentioned in the last section – the fear of a device failure leading to database failure. For Solaris, Linux or Windows users, it is especially easy to implement an SSD solution that is both fault tolerant and performant.

Ignoring SAN based solutions (to free up bandwidth/write cache for real data), the possible solutions include:

- SSD based storage array such as the Sun Storage F5100 flash fire array or Texas Memory Systems RamSan-620
- Backplane based such as the FusionIO ioDriveDuo
- Standard SAS/SATA RAID with SSD flash drives

The latter two are the cheapest and easiest to implement, but may be limited by OS support. The first may provide broader OS support as well as ability to share the SSD storage across multiple hosts.

Tempdb creation options

The way tempdb is created can also have an impact on performance. For years, DBA's have understood the impact of the 2MB allocation on the master device on tempdb performance and have attempted to eliminate it. There are other similar considerations – which may help performance in tempdb, depending on the workload and concurrency. The key creation options to consider are:

- In-Memory Tempdb
- Non-contiguous Chunk Allocations
- Splitting Log & Data

These are discussed in the following sections.

In-Memory Tempdb

As will be discussed later, one of the scalability features in ASE is the ability to have multiple tempdb's. One way to separate the workload is to bind processes performing low-latency time-critical OLTP functions to an in-memory tempdb vs. even a cached file system – thus reducing even the slightest possibility for physical IO induced waits for the SPIDs.

There are several advantages to this approach over even using a shared memory file system. A shared memory file system is a double hit on memory if striving to prevent physical IO's. First, of course, the shared memory file system must be created of the appropriate size. For example, let's use an arbitrary size of 10GB. At the OS level in Solaris (for example) you would do the following:

```
# create the mount point directory as normal
mkdir /mount_point

#change the directory owner to Sybase user
chown sybase /mount_point

# create the tmpfs region – this should be added to /etc/fstab
mount –F tmpfs –o size=10240 swap /mount-point

# verify it was mounted – consider adding to RUN_SERVER file
mount –v

# as sybase user, create a file in the file system to use as a database device
```

```
touch /mount-point/tmpfs_device.dat
```

On many OS's, you would also need at least 10GB of swap space in addition to other swapping requirements. At this point, all you have is the disk device that now Sybase ASE can use as a virtual device. Inside ASE, you would now do the more normal commands:

```
-- create the Sybase device
disk init
     name='tmpfs_device',
     physname='/mount-point/tmpfs_device.dat',
     size=10240,
     directio=false,
     dsync=false
go

-- create the temporary database
create temporary database tmpfs_tempdb
     on tmpfs_device=10240
go
```

At this point, we still have an issue. Any new page allocations or reads for this tempdb will occur in default data cache – or whichever cache it is bound to. To prevent SPIDs from incurring any physical IO's, you would need to create a cache big enough to totally contain the 10GB tempdb. Surprisingly, a 10GB cache is too small. Consider:

```
1> sp_helpcache '10G'
2> go
 644.04Mb of overhead memory will be needed to manage a cache of size 10G
(return status = 0)
```

Which means a cache of 10,885MB would be needed….or would it even be enough. Remember, select/into can use large memory pools – which means possibly creating a 16KB buffer pool. If that buffer pool is created too large or too small, it will cause a "physical IO". Remember too, that for cache pools greater than 300MB, the wash marker is set at 60MB – pages beyond the wash may be cleaned – so to be safe, we might need to add at least 60MB to the above which brings us closer to 11GB memory necessary for a 10GB caching requirement.

Net result: to fully cache a 10GB tempdb and use a shared memory file system, we would need ~21GB of memory.

We would also suffer the following:

- Buffer pool sizing for large buffer pool and log IO size buffer pool
- Cache contention as the normal MRU-LRU page chain would be enforced (unless using a relaxed cache strategy).
- Remembering to set "HK ignore cache" on the designated cache
- Cache partitioning considerations

By contrast, an in-memory tempdb using ASE 15.5's IMDB option does not have the external overhead. The similar process to the above would be:

```
sp_cacheconfig imdb_tempdb_cache, '10900M'          -- include cache overhead
go

disk init
     name='imdb_tempdb_dev',
     physname='imdb_tempdb_cache',
     size=10240,
     type='inmemory'
go

create inmemory temporary database imdb_tempdb
     on imdb_tempdb_dev=10240
go
```

The biggest advantage to this over the above is that in-memory databases in ASE both live and work in the cache in which they are created. Despite the fact that the cache is used like a device, ASE also knows it is a cache, and therefore does not "read the data into cache" in another cache such as the default data cache. Proof of this is visible in the following screen shot that shows the data cache contents after a select * is run in an IMDB.

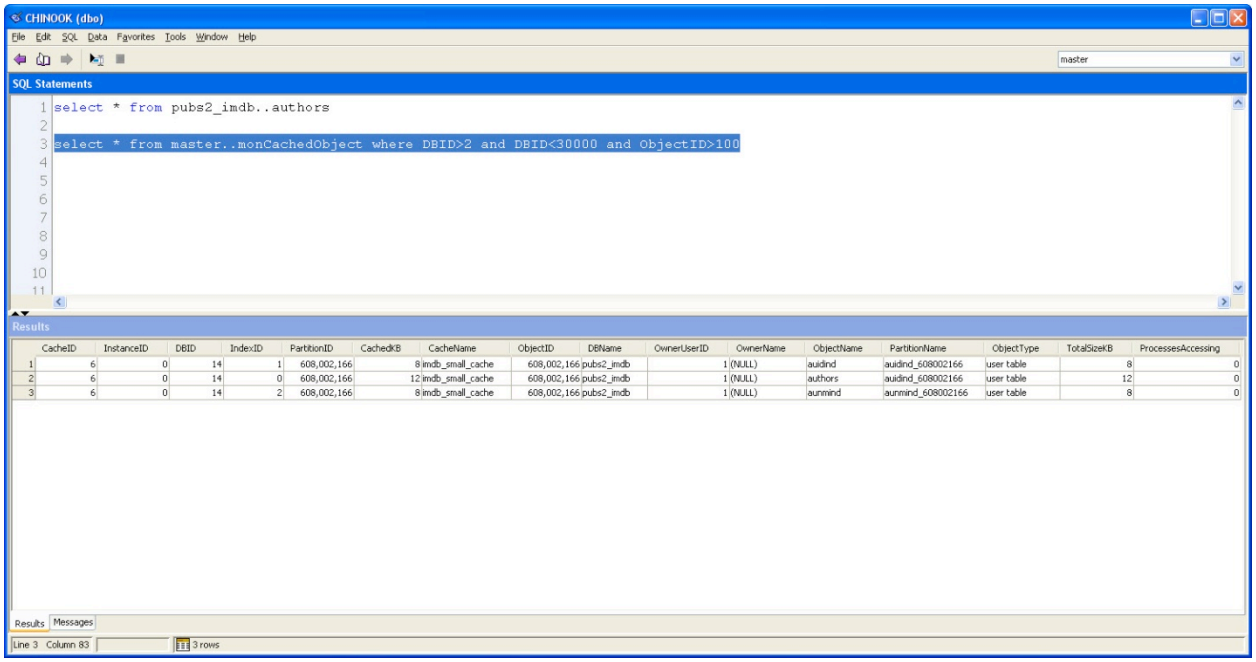


Figure 21 – The Single Memory Copy of IMDB vs. Shared Memory File Systems

The second advantage is that IMDB caches in ASE 15.5 are optimized for in-memory operations by eliminating the MRU-LRU chain, the wash area and even the need for different buffer pools.

The result is that an IMDB tempdb is much more optimal than an shared memory file system based tempdb.

Creating TempDB/Space Allocation

If not using an in-memory database, the space allocations can play a role in tempdb's when the tempdb can leverage multiple devices. Normally, DBA's create databases with fairly large fragments. The reason is to ease space management and to also ease the recreation of the database when loading a dump as the fragments need to be created in roughly the same sequence (only exception is adjacent like fragments can be combined). For tempdb's, this results in a database definition similar to:

```
create temporary database tempdb_02
    on tempdb_data01=5120,
       tempdb_data02=5120
    log on tempdb_log01=2048
```

However, it is actually be better to create the database using the syntax:

```
create temporary database tempdb_02
    on tempdb_sys01=100,
       tempdb_sys02=100,
       tempdb_sys03=100,
       tempdb_data01=100,
       tempdb_data02=100,
       tempdb_data03=100,
       tempdb_data04=100,
       tempdb_data01=100,
```

```
      tempdb_data02=100,
      tempdb_data03=100,
      tempdb_data04=100,
      ...
      tempdb_data01=100,
      tempdb_data02=100,
      tempdb_data03=100,
      tempdb_data04=100
   log on tempdb_log01=100,
      tempdb_log02=100,
      tempdb_log03=100,
      tempdb_log01=100,
      tempdb_log02=100,
      tempdb_log03=100,
      ...

exec sp_dropsegment 'default', tempdb_02, tempdb_sys01
exec sp_dropsegment 'default', tempdb_02, tempdb_sys02
exec sp_dropsegment 'default', tempdb_02, tempdb_sys03
exec sp_dropsegment 'system', tempdb_02, tempdb_data01
exec sp_dropsegment 'system', tempdb_02, tempdb_data02
exec sp_dropsegment 'system', tempdb_02, tempdb_data03
exec sp_dropsegment 'system', tempdb_02, tempdb_data04
```

Before we address the obvious panic over the number of fragments, the above shows 3 system segment devices, 4 data devices and 3 log devices being allocated in alternating 100MB chunks. This would work for a system based on raw devices, while a file system device implementation could actually use a separate file for each allocation chunk. The rationale for this is that ASE tends to fill one device before it spills to the other in a concatenated vs. striped fashion. Accordingly, in the first example, most of the IO's would go against tempdb_data01 and tempdb_log01. This exposes a couple of problems:

- Each device has its own pending IO queue with a device semaphore. If Sybase mirroring is enabled, IOs are delayed for configuration reasons, or other causes, the next process needing space has to wait for the semaphore.
- Additionally, certain IO operations - i.e. page splits in pre-15.0.2 servers - are synchronous operations (e.g. updates to #temp tables).

The main consideration is that dependent on the pagesize of your server, you will want to make sure that the allocation sizes are on even pagesize boundaries. For example, on a 2K server, a single OAM page for a 2K server may control 256 pages or 512K - anything to the even MB is fine. However, for a 16K server, the allocations may need to be in even multiples of 4MB (so 100 is fine). While less works, it just means there is an OAM page that is tracking fewer entries than it could.

Splitting Log & Data

By default, the system temporary database 'tempdb' is created with mixed data and log segments. For most production systems, this is sufficient as separating the two would have minimal performance gains. The combining of the log and data devices was done for ease of administration as DBA's shouldn't have to then worry about allocating sufficient log space - and because the log is truncated every checkpoint interval (60 seconds), the log space consumption should be fairly small.

So much for theory. The reality is that the mixed data and log segments on disk-based tempdb's lead to a number of availability issues as well as detract from overall performance on systems that used tempdb extensively. As discussed in more detail in the section on "Preventing Tempdb Log Suspends" later in this chapter, the biggest problem is that unexpectedly large data writes will often consume so much space that the transaction log is

often prevented from growing when it needs to. This leads to the usual log suspension in tempdb situation in which DBA's mistakenly assume it was a tempdb transaction log issue. As mentioned, that problem will be discussed later.

In addition to the above, prior to ASE 15.0.2 there was a bit of a performance hit with respect to tempdb and truncate log on checkpoint. If a database had truncate log on checkpoint enabled, with each checkpoint, the dirty pages would have to be flushed to disk. As mentioned earlier in the section on "Looking Under the Hood", dirty data pages are pinned to log records in the ULC. For ASE to write the dirty page to disk it needs to first write the log records to disk (write-ahead log). Hence, ASE will first write the log to disk- including flushing concerned ULC's log records to the transaction log. When the log is safely on disk, the checkpoint can continue with writing the dirty pages with the changes pinned to the log records just written. The impact of this is that since the checkpoint runs every 60 seconds and given a fairly active tempdb, a large number of IOs would be flushed to disk with each checkpoint in tempdb.

The net effect of both of these aspects is that especially for ASE 12.5, separating log and data segments for tempdb could improve overall performance - especially as this allowed the log segment to leverage file system or shared memory devices. Some Sybase DBA's prefer the separate log and data segments not only for the ability to leverage devices differently, but it provides a greater degree of control over the log filling/suspension problem.

For the system temporary databases, eliminating the mixed log/data segment issue involves a bit of tweaking. The process involves creating a different database and then swapping the sysdatabases entries. The steps are as follows:

1. Create a database called 'new_tempdb' with the appropriate devices sizes desired making sure that the first data space allocation is at least the same size as model - although 5-10MB is recommended. (see note at end of instructions)
2. Add the guest user to the database as well as enabling the usual tempdb database options: truncate log on checkpoint, bulkcopy, and delayed commit (optional)
3. exec `sp_configure 'allow updates', 1`
4. bcp out sysusages and sysdatabases (this step is for safety).
5. Shutdown the server
6. Add the boot time trace flag 3608 and single user mode to the RUNSERVER file by adding -M -T3608 to the command line. This trace flag boots the server recovering only the master database and the -M option starts it in single user mode (sa only). As a result, modifications to storage based system tables in master are fairly safe - but since most DBA procedures are in sybsystemprocs (such as sp_who), you will be limited to strict SQL
7. Execute the query "`select dbid from sysdatabases where name='new_tempdb'`". Remember the dbid from the output.
8. Execute the query "`select * from sysusages where dbid in (2,<dbid>)`" where <dbid> is the dbid value for new_tempdb from step #6. Remember the number of rows for each database.
9. Execute the following sequence:

```
-- SQL to swap the dbid's
begin tran
update sysdatabases set dbid=1000, name='old_tempdb' where dbid=2
```

```
update sysusages set dbid=1000 where dbid=2
update sysdatabases set dbid=2, name='tempdb' where dbid=<dbid>
update sysusages set dbid=2 where dbid=<dbid>
update sysdatabases set dbid=<dbid> where dbid=1000
update sysusages set dbid=<dbid> where dbid=1000
select * from sysdatabases where dbid in (2,<dbid>)
select * from sysusages where dbid in (2,<dbid>)
-- do NOT commit yet
go
```

10. Compare the output of the sysdatabases and sysusages with those prior to the modifications. If everything is as it should be (swapped), issue a 'commit tran'. Otherwise execute a 'rollback tran'.
11. Shutdown the server
12. Remove the -T3608 traceflag and -M (single user mode) from the RUNSERVER file
13. Restart the server
14. exec `sp_configure 'allow updates', 0`
15. Drop the new tempdb's data and system segments off of the master device by issuing:

```
use tempdb
go
exec sp_dropsegment 'system', tempdb, master
exec sp_dropsegment 'default', tempdb, master
go
```

Note: For several recovery as well as upgrade scenarios, it is critical that tempdb has a space allocation on the master device at least the same size as model - even if the allocation has no segments assigned to it (segmap=0). If necessary the segmap can be reset to 7 (log, default and system) and the server will boot in situations where a corrupt tempdb is blocking recovery. It is also critical that this be the first space allocation - with a lstart of 0 consequently it must be the first allocation when creating the database. If this is not done, recovery will be harder as you will need to manually calculate sysusages values which can fluctuate based on the server page size and may not be possible if the master device is full.

If the server fails to boot, you can recover by resetting the boot recovering master trace flag and use the bcp'd out data for sysusages and sysdatabases to fix the problem. However, this technique was published for years in the Sybase ASE Troubleshooting Guide, so it is fairly safe.

Multiple Tempdbs/Tempdb Groups

Probably one the most important features introduced early in ASE 12.5 - yet the most overlooked - is the ability to have multiple tempdb's and tempdb groups.

Multiple Tempdbs

Multiple tempdb's provide the following advantages:

- *Significant reduction of system catalog contention.* By using as few as 3-4 tempdb's, customers on high volume systems have noticed system catalog contention all but disappear.

- *Reduced log semaphore contention*. Although the session tempdb log cache helps significantly, there still can be considerable contention on the log semaphore in tempdb.
- *Increase Availability*. By having multiple tempdb's, if one tempdb's log fills causing transactions to suspend, users of other tempdb's are not affected.
- *Application Optimized Tempdb's*. Tempdb's can be created with optimal configurations for applications. For example, the normal user tempdb for worktables, etc. could be created using cached file systems, SSD's or even in-memory tempdbs, while tempdb's for reporting applications may simply use slower disks/SAN devices.
- *SA dedicated tempdb's*. Many of the system stored procedures use #temp tables. If only a single tempdb is available and it is full, commands such as sp_who and other system procedures necessary to try to troubleshoot the problem may not function. Note that the sa user (suserid=1) should never have a tempdb other than the default tempdb. For purposes of this discussion, we are referring to DBA's logging in with their own logins, but having sa_role.

Some may think that the first bullet is no longer applicable after ASE 15.0.2's Row Level Catalog implementation. While RLC does largely eliminate tempdb system table contention, the other reasons for multiple tempdb's still apply – and are still equally important. A typical high volume system should consider the following tempdb's:

- *OLTP Tempdbs.* One or more small tempdb's – preferably in-memory or minimally on a cached file system tuned for DBMS operations. If cached file system, the file system should be on a SSD or other high performance physical storage.
- *Time-Sensitive Batch/Report Tempdbs.* One or more large tempdb's – most likely on a cached file system with an underlying SSD
- *Less Critical Batch/Report Tempdbs.* One or more large tempdb's – typically on a cached file system tuned for DBMS operations but could be on slower devices such as tier 2 or tier 3.
- *Application Tempdbs.* These could be small or large depending on requirements – but tend to be smaller. The purpose often is to share data between multiple cooperative processes – for example those in a workflow process. The time criticality of the process will also dictate the type of device(s) for the tempdb – such as in-memory, SSD or tier 3 storage.
- *SysAdmin Tempdbs.* These need to be quite large and typically should be created using cached file system devices. The size is due to the fact that often DBA's have to move around large chunks of data if performing schema changes that exceed the available space of the original database.

Overall, using multiple tempdb's may require a bit of application design guidance. For example, developers who are used to creating shared temp tables via statements such as 'create table tempdb..<tablename>', the table will be created in the database specifically named 'tempdb'. While this might be viewed as optional for most sites, sites using ASE Cluster Edition will face this problem. For ASE Cluster Edition, each ASE instance has its own local system temporary

database (lstdb) when the usual query worktables and #temp tables are created. Additionally, a single global shared system temporary database 'tempdb' is created. The local temp databases have significant performance advantages in that they do not incur the overhead associated with the cluster (cluster locking, single log bottleneck, etc.) and can leverage local private devices not on the shared storage - which allows file system devices to be used for the local system tempdb's. As a result, applications being rehosted to an ASE cluster should be reviewed for possible hard-coding of the tempdb database name. The problem is that it will need to be hardcoded anyhow – but should be something resembling the application name – i.e. settlement_tempdb. Besides just being a good practice for standardization and to make it easy for identification, the other reason is that it is currently not possible to use an indirect reference via a variable such as the below:

```
-- None of these examples work – all return syntax errors at highlighted syntax
declare @tempdbname varchar(30)
select @tempdbname=db_name(@@tempdbid)
select * into @tempdbname..mytable from …
go

-- This also fails for the same reason
declare @temptable varchar(255)
select @temptable=db_name(@@tempdbid)+'..mytable'
create table @temptable (…)
go
```

The result is that application temporary tables that need to be shared across connections or sessions must use hardcoded temporary database names. While this restriction seems problematic, having a separate application temporary database dedicated to that application facilitates application transportability when managing workloads as will be discussed in a later section of this paper.

Tempdb Groups

In addition to multiple tempdb's, ASE also supports the notion of "tempdb groups". Most Sybase ASE DBA's are well aware of the fact that multiple temporary databases can be group together into a single tempdb group. Users and applications of that tempdb group are then distributed round-robin among the tempdb's within the group. For instance, by default, the database tempdb is in the 'default' tempdb group. As other tempdb's are added, they too are initially within this default group. Users connecting to the server would then be distributed in round-robin by connection to the tempdb's within this 'default' group - unless their application, or login specifies a separate tempdb binding. Part of the problem with using tempdb groups prior to ASE 15.5, however, was that with the exception of ASE/CE, ASE/SMP only really supported the 'default' group. While you could create multiple tempdb's, either they had to be a member of the 'default' group or they could only be individually bound to applications or logins. With ASE 15.5, ASE/SMP now supports the same level of multiple tempdb groups that was available in earlier releases of ASE/CE.

The concept of tempdb groups is important when establishing multiple tempdb's. The reason is that tempdb groups provide the logical ease of administration and application management to multiple tempdb's. For example, in the above section, one of the recommendations was to create multiple tempdb's for OLTP users – as well as multiple tempdb's for batch/reporting applications. We all can understand why the need for this – if we only created a single tempdb for all the OLTP users – we would likely still have a lot of tempdb contention among the OLTP applications within that tempdb – and any issues within that tempdb would immediately wreak

havoc on all the OLTP applications. Without tempdb groups we would one of two choices - either we would have to:

- customize the application to dynamically determine which tempdb to use for the session and use that tempdb in all statements submitted to the system, or …
- …arbitrarily alternate between the different tempdb's (hardcoded by name) for different instances when used with the application – trusting that with different users likely at different points of the application, that the tempdb workload would be distributed.

The first option makes application more difficult – especially for any server side code implementations in stored procedures or triggers. While it can be done using features such as dynamic SQL (the exec() function), the implementation can often get messy due to the different contexts between the calling routine and the statements executed by the call – especially in regard to #temp tables. While this has improved in recent ASE releases, there still is some aspects to consider. In the second case, scalability is threatened during business start, post-lunch and business end periods when users are often doing the same transactions concurrently – as well as the inflexibility for increasing workloads.

The answer, of course, is to simply create a tempdb group as logical view of the multiple tempdb's underneath and bind the applications to the group instead of the individual tempdb's. Note that when binding a tempdb, there are two possible values for the binding 'hardness': hard and soft. If the binding is set to 'hard', if the designated tempdb group/database is not available, the login fails. If the binding is set to 'soft', if the designated tempdb group/database is not available, the connection is assigned to 'tempdb'. This can be useful as a way of controlling logins for certain applications by forcing the database recovery order for application specific tempdb's to be after the recovery of the application database. For example, consider the following commands:

```
-- assume a single user database prod_db used for oltp & reporting
exec sp_tempdb 'create','oltp_temp_group'
exec sp_tempdb 'add','oltp_tempdb1','oltp_temp_group'
exec sp_tempdb 'add','oltp_tempdb2','oltp_temp_group'
exec sp_tempdb 'add','oltp_tempdb3','oltp_temp_group'
exec sp_tempdb 'bind', AP, 'oltp_app', GR, 'oltp_temp_group', null, 'hard'
exec sp_tempdb 'bind', AP, 'crystal reports', DB, 'dss_tempdb', null, 'hard'
exec sp_dbrecovery_order 'prod_db', 1
exec sp_dbrecovery_order 'oltp_tempdb1', 2
exec sp_dbrecovery_order 'oltp_tempdb2', 3
exec sp_dbrecovery_order 'oltp_tempdb3', 4
exec sp_dbrecovery_order 'dss_tempdb', 5
```

The net effect of these commands would be that OLTP users could start reconnecting after a server reboot as soon as the database 'prod_db' and at least one of the oltp_tempdb's were online while the Crystal Report™ users would have to wait for the dss_tempdb to come online before they could connect. A note of caution is that some DBA's prefer binding users with sa_role to an isolated DBA only tempdb. If this is used, users with sa_role should only ever use a 'soft' binding to avoid connection issues if the 'sa' account is locked. For further information, refer to the ASE Commands Reference for the stored procedures 'sp_tempdb' and 'sp_dbrecovery_order'.

Cluster Edition Local (Private) Tempdbs

ASE Cluster Edition has a few differences for tempdb that needs to be considered by DBA's. The first is that ASE/CE supports the notion of "local tempdb's" for each instance that can use "private devices". The second is that due to ASE/CE typically running on different hosts or instances, unique cache configurations are more difficult to support due to failover or load balancing workload migration.

Local tempdb's and Private Devices

The first and most obvious difference that an ASE/CE DB will notice is that each instance has its own private *local system tempdb* (lstdb). This lstdb is used for all #temp and work tables, while the global temporary database 'tempdb' is only used if the table is created explicitly with 'create table tempdb..<tablename>'. In addition to the one created during the initial cluster creation, additional local tempdb's can be added to the system. Using a local tempdb has a number of advantages:

- Clustering overhead such as cluster locking is avoided
- Single transaction log semaphore is avoided vis-à-vis the global tempdbs
- Private devices are supported

The last point is fairly important. Global databases such as tempdb must be created on shared storage in order for the device to be visible to all the instances. For ASE 15.0.x Cluster Edition, this is further restricted in that the shared storage must be created as raw partitions from the RAID LUNs without using a logical volume manager to further divide the device. As mentioned earlier, local storage or DASD devices may provide benefits in terms of eliminating IO bottlenecks and using cached file system devices may eliminate physical reads. Local tempdb's can use private devices which have none of the restrictions of shared storage - such as raw/SAN based platform. This is accomplished via an additional option on the disk init command, such as:

```
disk init
   name = <dev_name>,
   physname = <fullpath>,
   vdevno = <##>,
   size = <MB>,
   directio = false,
   dsync = false,
   instance = <instance>
```

Note that DIRECTIO and DSYNC are both set to false to ensure a cached file system device. The one problem is that currently during cluster creation time, private devices cannot be specified for lstdbs. As a result, it is recommended that rather than expanding the default lstdb to handle the local node's application workload, DBA's should instead create additional local tempdb's using private devices – relying on non-clustered file systems or local internal storage (such as SSD's) within the individual hosts.

Multiple tempdb's and Bindings

Similar to normal tempdb's, ASE Cluster Edition can have both local user temp databases as well as global temp databases. However, they are a bit harder to control.

Tempdb Bindings

First, applications and logins can only be bound to local tempdb's. The fun begins when you realize that with a single master database, you can't name the local tempdb's identically on all the instances - so when binding an application or login across the instances, it must be bound to different tempdb names. This is not quite as restrictive as it sounds from a local tempdb perspective as an application or login could be bound to different tempdb's per instance. As with ASE SMP, ASE CE can have tempdb groups. One change is that sp_tempdb can be thought of as creating tempdb's and groups in the *current* instance that the sa is connected to. Unbinding, however, must specify the instance name.

Similarly, the lack of ability to bind to a global tempdb is not as restrictive as it sounds either. Keep in mind that with a global temporary database, temp tables have to be created using the syntax 'create table <tempdb_name>..<tablename>' such as 'create table app1_tempdb..mytable'. As a consequence this explicit forcing of the catalog ensures the desired database is used for shared temporary tables. Remember that normal 'shared' temp tables in a global tempdb's are recoverable in the event of a node failure while #temp tables are not. As a result, such global temp tables should only be used for shared temp tables – a best practice whether using ASE/CE or simply the SMP version.

Tempdb Cache Bindings

The other area that can get confusing is the cache binding problem. Remember, earlier we recommended that some portion of memory be configured into a named cache for tempdb processing. The challenge is that we have different local tempdb's on different nodes as well as global tempdb's shared among the nodes – and depending on the application distribution across the nodes, some of the local tempdb requirements may be larger on some nodes vs. others. The general guidelines to consider are:

- Use local caches exclusively for local tempdb's.
- Use global caches for global tempdb's and application data caching requirements.

Where the different application requirements and workload distribution dictates larger local tempdb's, consider increasing the memory on that instance rather than decreasing the memory for global caches on that instance.

Tempdb Recommendations & Best Practices

The following best practices are recommended for tempdb:

- Increase session tempdb log cache size to accommodate most OLTP tempdb requirements.
- Use multiple tempdb's to separate application workloads such as OLTP, reporting and batch processing.
- Use multiple tempdb's for each workload (such as OLTP) to provide scalability – but group them into a single tempdb group for manageability.

- For OLTP or time-critical processing applications, use either an in-memory tempdb, or a tempdb that is created on a cached file system using an underlying high-speed device such as a Solid State Disk.
- Do not block replicate a tempdb device – if tempdb devices are created on SAN logical LUNs, keep the LUNs separate from physical devices that are block replicated if the block replication is at the physical device vs. logical LUN level.
- Try to avoid using SAN devices for tempdb altogether if possible. While a SAN likely supports the best throughput, tempdb generally is not recoverable and inflicts considerable IO load on the SAN, the SAN write cache and other assets that are best left to the real business data. Use an internal high speed device such as an SSD or a less expensive (vs. SAN) RAID set with a decent amount of cache.
- If using a file system device, make sure of the following:
  - ✓ The file system supports asynchronous IO
  - ✓ The file system is tuned and IO driver is optimized
  - ✓ Sufficient host memory is available for file system caching and that ASE will not be paged/swapped if the caching expands.
- Instead of using tmpfs or other shared memory file system, use an in-memory tempdb
- Use separate application specific tempdbs for 'shared' tempdb tables (vs. #temp or worktables). For ASE/CE bind application tempdb's to global caches.
- If using multiple physical disk devices for tempdb, consider allocating the space in chunks to try to distribute the IO load. However, use a RAID implementation first – only do this on top of RAID or non-RAID devices if necessary.
- Use private devices for local tempdb's. This allows file system and DASD devices for performance considerations. Bind them to local caches vs. global caches.
- Use separate tempdb's for system administration activities.

Enabling Virtualization: Process Management

Without some form of intelligent policy based controls, it is very easy for a process or class of users to monopolize a system or at least some of the resources. Sybase provides several different mechanisms for controlling processes to ensure SLA's are achievable. In this section, we will take a look at several that control individual process resource usage that can be important when constraining users with similar profiles as well as separating more critical processes.

Engine Groups & Dynamic Listeners

Engine groups and dynamic listeners are actually two different features that are more than complementary. In fact, while they can be used individually, using one without the other often leads to undesired consequences. In addition a typical implementation also requires using execution classes – a topic covered in more detail in the next section.

Engine Groups

One of the oft overlooked and yet one of the most critical features for preventing system monopolization by particular users or applications is the concept of engine groups. Similar in one context to OS constraints on processors (such as Solaris's psrset command), engine groups are used to restrict different users/applications to groups of ASE engines – reserving the others for other users of the system. Some of the more common uses for engine groups include:

- Restricting internet users to just a small subset of the available engines to prevent surges in internet usage from impacting overall system performance for internal users.
- Isolating different applications to prevent one from monopolizing the system
- Isolating engines for dedicated processes (such as the replication agent or message bus interfaces) guaranteeing low latency for those processes.

Consider the following commands:

```
-- add an engine to an engine group or create the engine group if it doesn't exist
sp_addengine engine_number, engine_group [, instance_id]

-- remove an engine from an engine group
sp_dropengine engine_number [,engine_group] [,instance_id]
```

Note: these commands should not be confused with sp_engine – which brings an engine online or offline. The above commands do not affect the engine's running state. By default, there are two system predefined engine groups: ANYENGINE and LASTONLINE. These predefined groups cannot be modified, consequently to fully separate two different applications, you need to create two different engine groups covering distinctly separate engines and bind the two applications to the new groups. If one of the applications is not bound, the result is that by default it is still bound to the ANYENGINE group – which means it could still contend for cpu resources on the same engines as those covered by the other engine group.

Creating and binding users to an engine group is really a four step process. First, you have to create the engine group using the above commands. The second step is to create an execution class that the application will use. The third step is to bind applications or logins to the new execution class. The fourth and final step is to implement dynamic listeners in such a way to ensure complete process separation. For example:

```
-- create an engine group on engines 0 & 1 to handle message traffic
exec sp_addengine 0, 'MSGEngineGroup'
exec sp_addengine 1, 'MSGEngineGroup'
go

-- create an engine group on engines 2-8
exec sp_addengine 2, 'OLTPEngineGroup'
exec sp_addengine 3, 'OLTPEngineGroup'
exec sp_addengine 4, 'OLTPEngineGroup'
exec sp_addengine 5, 'OLTPEngineGroup'
exec sp_addengine 6, 'OLTPEngineGroup'
exec sp_addengine 7, 'OLTPEngineGroup'
exec sp_addengine 8, 'OLTPEngineGroup'
go


-- create an engine group to handle internet users on engines 9-12
exec sp_addengine 9, 'WWWEngineGroup'
exec sp_addengine 10, 'WWWEngineGroup'
exec sp_addengine 11, 'WWWEngineGroup'
exec sp_addengine 12, 'WWWEngineGroup'
go

-- create an execution classes to use with the engine groups
exec sp_addexeclass 'MSGExeClass', 'HIGH', 0, 'MSGEngineGroup'
exec sp_addexeclass 'OLTPExecClass', 'MEDIUM', 0, 'OLTPEngineGroup'
exec sp_addexeclass 'WWWExecClass', 'MEDIUM', 0, 'WWWEngineGroup'
go

-- bind applications to the exe classes just created
exec bindexeclass 'msgbus', 'lg', null, 'MSGExeClass'
exec bindexeclass 'ctisql', 'ap', null, 'OLTPExecClass'
exec bindexeclass 'isql', 'ap', null, 'OLTPExecClass'
exec bindexeclass 'webuser', 'lg', null, 'WWWExecClass'
go

-- start a listener for the messaging application
exec sp_listener 'start', 'monsterlinux:31001', '0,1'
go

-- start a listener for the OLTP applications
exec sp_listener 'start', 'monsterlinux:31002', '2-8'
go
-- start a listener for the web applications
exec sp_listener 'start', 'monsterlinux:31003', '9-12'
go
```

You may have noticed one slight problem with this workload management implementation – unless the application or login name is explicitly specified, any login not listed will have an execution class of EC2 (the default) and engine group of ANYENGINE (the default) – in other words, uncontrolled. The way to think of using engine groups is that engine groups ***restrict*** applications to a subset of the available engines. Applications/users not bound to an engine group are therefore unrestricted and have access to any engine – which can lead to problems without continued monitoring and adjustment of engine group members.

For systems using actual login names due to security requirements, the aspect of using login names is daunting. On the other hand, it is extremely difficult controlling application names – especially in 3-tiered systems in which a common connection pool is used for multiple applications (a classic implementation flaw of many middle tier connection pools). Unfortunately, there is no way to control the default engine group or execution class – so – the result is that using engine groups is often a "best effort" implementation followed by regular monitoring to adapt to changing applications and users.

Experience has shown that when using 12 or more engines and 4 or more different applications or classes of users, engine groups resolve a lot of resource contention within the system. Of course, like any workload management interface, the general idea is to use as few as necessary to accomplish the task as too many could result in application starvation for CPU on some engine groups. Generally, you should keep the number of engine groups per server to less than 6 to prevent contention for process execution. The trick can be determining the initial number of engines that may be appropriate for each engine group. One method is to use the MDA tables to measure the amount of CPU used by the various applications – considering of course that one of the applications may be misbehaving already and consuming too much CPU. Consider the following MDA tables:

```
create table monProcess (
        SPID                        int,
        InstanceID                  tinyint,
        KPID                        int,
        ServerUserID                int,
        BatchID                     int,
        ContextID                   int,
        LineNumber                  int,
        SecondsConnected            int,
        DBID                        int,
        EngineNumber                smallint,
        Priority                    int,
        FamilyID                    int NULL,
        Login                       varchar(30) NULL,
        Application                 varchar(30) NULL,
        Command                     varchar(30) NULL,
        NumChildren                 int NULL,
        SecondsWaiting              int NULL,
        WaitEventID                 smallint NULL,
        BlockingSPID                int NULL,
        BlockingXLOID               int NULL,
        DBName                      varchar(30) NULL,
        EngineGroupName             varchar(30) NULL,
        ExecutionClass              varchar(30) NULL,
        MasterTransactionID         varchar(255) NULL,
)
go


create table monProcessActivity (
        SPID                        int,
        InstanceID                  tinyint,
        KPID                        int,
        ServerUserID                int,
        CPUTime                     int,
        WaitTime                    int,
        PhysicalReads               int,
        LogicalReads                int,
        PagesRead                   int,
        PhysicalWrites              int,
        PagesWritten                int,
        MemUsageKB                  int,
        LocksHeld                   int,
        TableAccesses               int,
        IndexAccesses               int,
        TempDbObjects               int,
        WorkTables                  int,
        ULCBytesWritten             int,
        ULCFlushes                  int,
        ULCFlushFull                int,
        ULCMaxUsage                 int,
        ULCCurrentUsage             int,
        Transactions                int,
        Commits                     int,
        Rollbacks                   int,
)
go
```

By monitoring both tables together and then aggregating the data by application or user login you can get a decent picture of the amount of CPU time used per application or login. By comparing this to the total CPU usage, you can get an idea of how many engines to use as a starting point. Consider the following queries:

```
-- assume MDA data is collected into a repository and includes a SampleTime column for each
-- sample of MDA data.  Assume the repository table name takes the form <MDA_table>_hist.
select SampleTime, TotalCPU=sum(CPUTime)
  into #total_cpu
  from monProcessActivity_hist
  group by SampleTime

select mp.SampleTime, mp.EngineGroupName, mp.Application, ApplicationCPU=sum(mpa.CPUTime)
  into #application_cpu
  from monProcessActivity_hist mpa, monProcess mp
  where mp.SPID=mpa.SPID
    and mp.KPID=mpa.KPID
  group by mp.SampleTime, mp.EngineGroupName, mp.Application
go
```

Using the MDA tables is critical in environments where engine groups overlap. This could be done for a variety of reasons such as for nightly batch processes vs. daily OLTP – and in-fact exists by default do to the existence of the default execution classes EC1,EC2,EC3 and the pre-defined engine groups for those execution classes.

Dynamic Listeners

The problem with engine groups is that they only control which engines are used for process execution. Each user spid also has an associated network engine which is assigned at connection time by the TCP stack based on network load/response time for the socket port listening on. As a result, even though a spid may be blocked from executing on an engine, it may need to have access to an engine if it is that spid's network engine. Consequently, by themselves, engine groups are not very optimal without dynamic listeners to control the network engine for different users.

Note that dynamic listeners really is meant for non-Microsoft Windows platforms. On Microsoft Windows, ASE does not use the normal concept of OS processes for engines, but rather a single OS process for the instance and threads for the engines as well as IO processing. As a result, on Microsoft Windows, dynamic listeners do not work and all users connect to engine 0 (as it is the only process available to handle network tasks).

Dynamic listeners infers that ASE can dynamically start and stop network listener threads on individual socket ports and individual threads. This is controlled via the sp_listener stored procedure:

```
-- start a new listener:
exec sp_listener 'start', 'hostname:port', '<list of engines>'
go

-- stop a listener on a specific list of engines
exec sp_listener 'stop', 'hostname:port', '<list of engines>'
go


-- stop a listener on all engines….must be running on all engines.
-- if the listener is not running on some engines, the below
-- command is likely to fail.
exec sp_listener 'stop', 'hostname:port'
go

-- stop a listener on all engines for which a listener is running on.
-- unlike the previous example, this command will not fail if a
-- listener is not running on any given engine
exec sp_listener 'stop', 'hostname:port', 'remaining'
go


-- get a list of listeners currently active
exec sp_listener 'status'
go

-- get a list of listeners currently active on a particular port
```

```
exec sp_listener 'status, 'hostname:port',
go

-- suspend a listener from accepting new connections
exec sp_listener 'suspend', 'hostname:port', '<list of engines>'
go

-- resume accepting new connections on a particular listener
exec sp_listener 'resume', 'hostname:port', '<list of engines>'
go
```

While the syntax does support the notion of a keyword 'remaining' in place of the list of engines, in practice, it can be hard to use without thinking about what it infers. The reason is that is from the context of the listener on that hostname and port combination. In other words, if engines 0-2 already are listening on port 30000 for a 6 engine host "monsterlinux", the following command starts the listener on engines 3-5:

```
exec sp_listener 'start', 'monsterlinux:30000', 'remaining'
go
```

Similarly, if a network listener was only running on engines 3-5, but not running on engines 0-2 or 6-11 (assuming 12 engine server), the following command stops the network listener only on engines 3-5:

```
exec sp_listener 'stop', 'monsterlinux:30000', 'remaining'
go
```

But what happens if the following sequence is issued (assume 13 engines online):

```
-- start a listener for the messaging application
exec sp_listener 'start', 'monsterlinux:31001', '0,1'
go

-- start a listener for the OLTP applications
exec sp_listener 'start', 'monsterlinux:31002', '2-8'
go
-- start a listener for the web applications
exec sp_listener 'start', 'monsterlinux:31003', 'remaining'
go
```

The answer is a bit confusing as 'remaining' in this case would be misconstrued as the remaining engines 9 to 12. However, it is from the context of the listener on port 31003, in which case it would start a listener on that host and port on any engine for which a listener was not already running. In this case, it would likely start a listener on all engines. As a result, remember that the 'remaining' keyword only applies to the listener in the current command. A best practice is to simply ignore the 'remaining' keyword and use the explicit engine list or only use the 'remaining' keyword when suspending, resuming or stopping a network listener.

The suspend/resume commands would be an interesting method of slowly degrading a system in preparation for off-lining the applications for system maintenance. Some time prior to the maintenance period, you could 'suspend' the listeners used by that application to prevent additional logins while allowing those already connected to complete their work. However, you cannot suspend all the listeners – even if connected. This prevents a situation in which you cannot reconnect to the server if you get disconnected due to an error. If needing to suspend all the current active listeners, you should first start a listener on any unused port, connect over that port and then suspend the others. The connection error message for a suspended listener is different from when the server simply isn't running:

```
-- attempt to connect to server not running:
C:\work>isql -Usa -SYUKON
Password:
CT-LIBRARY error:
        ct_connect(): network packet layer: internal net library error: Net-Lib protocol driver call
to connect two endpoints failed
```

```
-- attempt to connect to a suspended listener
C:\work>isql -Usa -SCHINOOK
Password:
CT-LIBRARY error:
        ct_connect(): user api layer: internal Client Library error: Read from the server has timed
out.
CT-LIBRARY error:
        ct_connect(): network packet layer: internal net library error: Net-lib operation timed out

C:\work>
```

Note that when suspending a listener, existing connections are not affected. So, until they log off, applications remain connected and can submit queries. If preparing for maintenance, you will need to have another process that kills idle connections and then finally kills active non-sa connections at the specified time. As a result, the best use of the suspend/resume listener commands is to react to uncontrolled (or poorly controlled) applications that may be blocking other users from accessing the system.

The best technique for using dynamic listeners in managing workload is to simply start an ASE on a system administration only used port that is not published to general users. Then, once the server is fully recovered and any post-recovery tasks completed, the system can be brought online simply by starting the network listeners on the application defined ports. This was illustrated in the earlier example with the lines:

```
-- start a listener for the messaging application
exec sp_listener 'start', 'monsterlinux:31001', '0,1'
go

-- start a listener for the OLTP applications
exec sp_listener 'start', 'monsterlinux:31002', '2-8'
go
-- start a listener for the web applications
exec sp_listener 'start', 'monsterlinux:31003', '9-12'
go
```

The local interfaces/sql.ini file would only have the original starting port numbers (i.e. 30000) and that entry would only be in the local interfaces as well as the interfaces/sql.ini for the host machines used by the DBA's. This can be a bit tricky as often applications may run on the same host as the server. There are several methods for dealing with this.

- Only add query lines for the application ports
- Use a separate interfaces file for the server and add a –I startup parameter

With respect to the former, it would be possible to create an interfaces file similar to:

```
YUKON
        master tcp ether 127.0.0.1 30000
        query tcp ether 127.0.0.1 30000
        master tcp ether 192.168.1.90 30000
        query tcp ether 192.168.1.90 30000
        master tcp ether 192.168.1.91 30000
        query tcp ether 192.168.1.91 30000
        query tcp ether 192.168.1.90 30001
        query tcp ether 192.168.1.90 30002
        query tcp ether 192.168.1.90 30003
        query tcp ether 192.168.1.91 30001
        query tcp ether 192.168.1.91 30002
        query tcp ether 192.168.1.91 30003
```

With the intention that the physical NIC ports 30001-30003 would be used by applications. In this case, the server YUKON would boot and using the 3 'master' lines only start listeners on port 30000. You could later start listeners using sp_listener on the other ports for the specific

engines desired. However, a better approach is to use the concept of logical server names as in:

```
YUKON
        master tcp ether 127.0.0.1 30000
        query tcp ether 127.0.0.1 30000
        master tcp ether 192.168.1.90 30000
        query tcp ether 192.168.1.90 30000
        master tcp ether 192.168.1.91 30000
        query tcp ether 192.168.1.91 30000

SALES
        query tcp ether 192.168.1.90 30001
        query tcp ether 192.168.1.91 30001

CUST_SVC
        query tcp ether 192.168.1.90 30002
        query tcp ether 192.168.1.91 30002

SUPPORT
        query tcp ether 192.168.1.90 30003
        query tcp ether 192.168.1.91 30003
```

The effect is the same, however, as we will be later discussing, this provides an element of transportability for workload management in virtualized environments as well as prepares applications for cluster deployments using ASE/Cluster Edition.

In summary, in order for the engine group example earlier to work as desired, we would need the following interfaces file or sql.ini equivalent distributed to the client machines or middle tier servers:

```
APP_MSGBUS
        query tcp ether monsterlinux 30001

APP_OLTP
        query tcp ether monsterlinux 30002

APP_WEB
        query tcp ether monsterlinux 30003
```

By specifying the correct logical server names from the connection pools or applications, the application or user connection is automatically directed to the correct network engines as well as processing engines. This shows how application workload for a single application can be managed. Multiple applications are simply an extension of this technique.

Like all resource assignments, you need to be careful with network listeners. The maximum number of network listeners is 32. However, this is further restricted by the configuration parameter ‘max number network listeners’, which defaults to 5. Note that by a ‘network listener’, we are referring to a host:port pairing and not to individual engine instances of the listener. However, there is no direct association between interfaces/sql.ini and network listeners. Consequently, if you have 100 applications on a single large ASE instance, you could define 100 logical server names in the interfaces file spread across the various network listeners that you have defined. Applications that share the same data in a read/write mode and share the same engines in the same or overlapping engine groups should use the same network listeners.

Execution Classes

The second mechanism is ‘execution classes’ – which was also used in the above engine group example. Generically, an execution class is a way of establishing the relative priority of process execution.

Creating Execution Classes

ASE ships with three predefined execution priorities: LOW, MEDIUM, and HIGH – and with three corresponding predefined execution classes: EC3, EC2 and EC1. You can create your own custom execution class via the stored procedure sp_addexeclass:

```
-- create a new execution class

-- timeslice is currently ignored
sp_addexeclass <classname>, [HIGH | MEDIUM | LOW], [<timeslice> | NULL], engine_group [, instance_id]


-- drop an existing execution class
exec sp_dropexeclass <classname>

-- bind an application to an execution class
sp_bindexeclass <object_name>, [ap | lg | pr], <scope>, <classname>

-- show a list of all execution classes
exec sp_showexeclass

-- show what is bound to a particular execution class
exec sp_showexeclass <classname>

-- unbind an application from an execution class
sp_bindexeclass <object_name>, [ap | lg | pr], <scope>
```

As we are all well aware, an application, login and stored procedure all could be bound to different execution classes….and confusingly have different execution priorities. The execution priority as documented depends on either the precedence rule if the bound objects are different (in this case app, login & proc are different) or if multiple bindings exist, the server uses the scope rule.

- Precedence rule: procedure → login → application
- Scope rule: narrowest scope wins

The last one takes a bit of puzzlement, but consider the following example:

```
-- bind the isql application to EC3 as it is called by a lot of reports
exec sp_bindexeclass 'isql', 'ap', NULL, 'EC3'
go

-- bind the isql application for sa user to EC1
exec sp_bindexeclass 'isql', 'ap', 'sa', 'EC1'
go
```

Even though the first binding affects all invocations of isql, because the sa user has a narrower scope binding, when the sa connects, the sa user will have a priority of EC1.

If there are only three execution priorities and there are already three execution classes, the first question that might be asked is why or when would you create separate execution classes from the default. Technically, it is true that creating a separate execution class via 'exec sp_addexeclass "FastLane", "HIGH"' is no different than simply using the system provided EC1 execution class at a priority level. However, a key point to remember is that an execution class really controls *two* aspects:

- The execution priority
- The engine group to use

The last bullet is the key difference and the oft overlooked aspect. The default execution classes are defined as:

System Exe Class	Priority	Engine Group
EC1	HIGH	ANYENGINE

EC2	MEDIUM	ANYENGINE
EC3	LOW	LASTONLINE

Remember, as processes, applications, stored procedure executions or logins can only be bound to execution classes. Since you cannot change an engine group for an execution class (other than unbinding and rebinding) and system defined engine groups cannot be altered, this means that in order to use custom defined engine groups, you must also custom define execution classes that are defined to use those engine groups.

In addition to binding processes to execution classes, individual processes can be adjusted using the stored procedure sp_setpsexe:

```
-- adjust the priority of a single spid
exec sp_setpsexe <spid>, 'priority', [HIGH | MEDIUM | LOW]

-- move a process into an engine group
exec sp_setpsexe <spid>, 'enginegroup', <enginegroup>

-- clear a processes priority
exec sp_clearpsexe <spid>, 'priority'

-- clear a processes engine group setting set with sp_setpsexe
exec sp_clearpsexe <spid>, 'enginegroup'

-- show process control information
exec sp_showcontrolinfo [<object_type> [, <object_name>] [, <spid>]]
```

Note that earlier we defined only a single priority for each execution class in our example:

```
-- create an execution classes to use with the engine groups
exec sp_addexeclass 'MSGExeClass', 'HIGH', 0, 'MSGEngineGroup'
exec sp_addexeclass 'OLTPExecClass', 'MEDIUM', 0, 'OLTPEngineGroup'
exec sp_addexeclass 'WWWExecClass', 'MEDIUM', 0, 'WWWEngineGroup'
go
```

The obvious question is what happens. In the current ASE implementation, in addition to internal run queues/priorities, there are only 3 user level server wide run queues plus 3 user level run queues for each engine. Typically tasks are on the three server run queues as the engine run queues are only used when only that specific engine has been assigned to run that task. An example of this is network IO. Each run queue corresponds to the 3 priorities HIGH, MEDIUM, and LOW. By default, users are bound (if you will) to the EC2 execution class – which means that they run at a MEDIUM priority. Priority in this context does not affect the number of CPU cycles, but just which run queue they are put on when woken up from a sleep (due to network, blocking or physical IO) or when the timeslice is exhausted and they simply go back on the run queue. As a result, a user with a LOW priority may actually use more CPU time than one with a HIGH priority – especially if the LOW priority user is performing a lot of in-memory table scans while the HIGH priority user is doing a lot of write activity (and being put to sleep each time as a result).

When interrogating the run queues, each engine checks the engine local run queue for the priority prior to the server global run queue. Consequently, the order is something like:

```
engine HIGH → server HIGH → engine MEDIUM → server MEDIUM → engine LOW → server LOW
```

This also explains an confusing aspect with process escalation and blocking. A common occurrence is that DBA's will notice that a particular spid is blocking others – and that spid itself is sleeping (either awaiting IO or itself blocked by another user). However, the process will have been escalated to a HIGH priority. The first thought in most DBA's minds is that this is a mistake – a process that is causing contention (in their minds) should not be HIGH priority. But, remember, the process is currently ***sleeping*** as it itself is blocked or awaiting IO. Once the

process is no longer blocked or the IO returns, it will be woken up and put on the HIGH priority queue – meaning it will get to execute quicker, thus eliminating the contention faster.

Generally speaking, most of the users on the system should be average priority or MEDIUM priority. In our earlier example, we had defined the OLTP and Web users as MEDIUM priority while the messaging system users were HIGH priority. In one sense, when using totally isolated engine groups, it really doesn't matter as the engine groups will deconflict the different users. However, even for a messaging system, you may find out that some message queues are of a higher priority than others. The same situation for the other engine groups – some users might need higher priority. In that case, the best definition for our execution classes in our example should have been:

```
-- create an execution classes to use with the engine groups
exec sp_addexeclass 'MSG_HIGH_EC', 'HIGH', 0, 'MSGEngineGroup'
exec sp_addexeclass 'MSG_MED_EC', 'MEDIUM', 0, 'MSGEngineGroup'
exec sp_addexeclass 'MSG_LOW_EC', 'LOW', 0, 'MSGEngineGroup'
exec sp_addexeclass 'OLTP_HIGH_EC', 'HIGH', 0, 'OLTPEngineGroup'
exec sp_addexeclass 'OLTP_MED_EC', 'MEDIUM', 0, 'OLTPEngineGroup'
exec sp_addexeclass 'OLTP_LOW_EC', 'LOW', 0, 'OLTPEngineGroup'
exec sp_addexeclass 'WWW_HIGH_EC', 'HIGH', 0, 'WWWEngineGroup'
exec sp_addexeclass 'WWW_MED_EC', 'MEDIUM', 0, 'WWWEngineGroup'
exec sp_addexeclass 'WWW_LOW_EC', 'LOW', 0, 'WWWEngineGroup'
go

-- bind the high priority message queue login to correct exec class
exec sp_bindexeclass 'hiqh_mq', 'lg', NULL, 'MSG_HIGH_EC'
go
```

So, in a sense, you really need to create 3 execution classes each time – one for each priority.

Users & Priorities

When planning such a system, the next step is to determine:

- Which users/applications could benefit from an increased priority
- Which users/applications will cause harm if their priority is increased

To understand the possible impacts, it helps to understand the sleep/runnable/running cycle:

- A process is put to sleep while waiting physical IO, a lock, or network IO
- Once woken, a process is put on the run queue (runnable state) for the priority it has based on execution class or sp_setpsexe settings
- Once running, a task will execute until it needs to do physical IO, a lock or network IO (on different engine) and therefore is put to sleep – or the timeslice expires

From this we can see the following characterizations:

- Processes that do a lot of really short physical IO (reads or writes) can benefit from a higher priority. They will not monopolize the CPU as a result due to frequent sleep cycle while waiting IO.
- Highly contentious processes that do short IO would also benefit from a higher priority. For example, the sequential key table problem frequently mentioned in this paper.
- Processes that do a lot of cached reads – especially longer queries – could monopolize the CPU as they will simply move from running to runnable and back again.

- Processes that do quick/short reads could benefit from a higher priority – especially if executing on a engine other than their network engine.

What about long running reports that do a lot of physical IO? Obviously, raising the priority can help it finish quicker with concurrent users on the system assuming the bulk of the user activity is of lower priority. The question is should such a report be favored? Unless it is causing a lot of contention (e.g. isolation level 3), the answer likely is "no". If not, then should it have a low priority? If the physical IO's it is doing are causing hardware resource contention, then possibly so long as it is not causing any logical contention with locking. As you can see, the real situation is that while it might be tempting to automatically assign a low priority to long running reports that are doing a lot of physical reads, the priority really shouldn't be changed unless there is a problem. At that is true for most tasks, which avoids the nightmare of trying to manage every task individually.

There are several classes of users, however, that it makes sense to assign a HIGH priority by default:

- RS maintenance users
- Message system users
- Automated system users (e.g. Stock Order Entry or Call Center systems) with short queries/DML.
- Stored procedures that generate the next sequential key from key tables

The last one might be a bit of a surprise. Typically, a common sequential key stored procedure will resemble:

```
create table key_table (
        keynum          int                     not null,
        cur_spid int    default 0       not null,
                primary key (keynum)
)
go

create proc get_next_key  @nextkey int output
as begin
        begin tran
                update key_table
                   set keynum=keynum+1,
                         cur_spid=@@spid
                select @nextkey = keynum from key_table
        commit tran
return 0
end
```

No matter what the priority of the user is who executes this procedure, they will cause blocking as all users need to be serialized through this code. Since the user is in a transaction, a physical log write must occur – and the user holding the lock is put to sleep until the log write completes. If the user remains at the default MEDIUM priority, other tasks not dependent on the lock will also be in the MEDIUM run queue. The result is that even after the log write completes, the user may be waiting on CPU before releasing the lock – thus extending the time contention is occurring. This can be resolved somewhat via binding the procedure to a higher priority:

```
-- increase the priority of the next key procedure to reduce blocking time
use <database>
go
exec sp_bindexeclass get_next_key, 'pr', 'dbo', 'MSG_HIGH_EC'
exec sp_bindexeclass get_next_key, 'pr', 'dbo', 'OLTP_HIGH_EC'
exec sp_bindexeclass get_next_key, 'pr', 'dbo', 'WWW_HIGH_EC'
exec sp_bindexeclass get_next_key, 'pr', 'dbo', 'EC1'
go
```

```
exec sp_showcontrolinfo 'pr', 'get_next_key'
go
```

Notice that we had to raise the priority for the procedure across multiple execution classes – raising it for the default execution class EC1 will not help the users assigned to the engine groups via the other execution classes in our sample system. This is where sp_showcontrolinfo is really useful as it can be used to verify the settings across all the execution classes. This can be done whether the object is bound or not – if it is not bound, you will simply get a null result set from sp_showcontrolinfo.

While it might also be tempting to simply lump non-critical (e.g. web vs. internal) users into the LOW priority, a better solution is to simply set up an engine group and isolate the tasks completely vs. trying to manage with priorities.

Resource Governor & Time Ranges

In addition to execution classes and engine groups, ASE also supports a resource governor which can restrict query execution based on time limits, result set size or other limitations. In recognition that different applications may have different restrictions based on time of day, these resource limits may additionally have time ranges for enforcement.

Resource Limits

Both engine groups and execution classes establish relative control parameters for application execution. Definitive control can be establish through the use of resource limits enforced by the resource governor. Currently, the following resource limits can be imposed:

Limit Type	Description
row_count	The number of rows *returned* by a query.
elapsed_time	The wall clock time in seconds that a query, batch or transaction can run.
io_cost	The actual cost or the optimizer's cost estimate for processing a query (note this is *IO cost* not number of IO's)
tempdb_space	Limits the number of *pages* of tempdb a single query, transaction or batch can consume

The limits can be defined for an application or login using the command:

```
exec sp_add_resource_limit <loginname>, <appname>, <timerange>, <limittype>, <value>,
        <enforced>, <action>, <scope>
```

The time range for enforcement has to be a predefined time range name (see next discussion). The values for how to enforce depends on the type of limit being enforced:

Enforce	Limit Type	Description
1	io_cost	Action is taken when the *estimated* I/O cost of execution exceeds the specified limit.
2	row_count elapsed_time io_cost	Action is taken when the *actual* row count, elapsed time, or I/O cost of execution exceeds the specified limit.
3	io_cost	Action is taken when either the *estimated* cost *or* the *actual* cost exceeds the specified limit.

Note that as documented, the enforced attribute is a bitmask in that an enforced attribute=3 is a combination of 1 and 2.

The ‘action’ parameter also has a similar decode table:

Action Code	Description
1	Issues a warning (message) to the client application
2	Aborts the query batch with an exception
3	Aborts the transaction with an exception
4	Kills the session

The decode table for ‘scope’ is:

Scope Code	Limit type	Description
1	io_cost row_count	Query
2	elapsed_time	Query batch (one or more SQL statements sent by the client to the server) in a single command batch.
4	elapsed_time	Transaction
6	elapsed_time	Query batch and transaction

Notice that scope also is a bitmask as 6 includes both 4 and 2.

In the past, resource limits were often used to find where long running queries originated from by simply adding the warning action code. How well this worked depended on exception logging in the application. With the introduction of QP Metrics in ASE 15, this is no longer necessary as long running queries can be identified and logged much easier than relying on application exception logging when a warning was returned.

The guiding principal of resource limits is to prevent a single query batch or transaction from monopolizing all the resources. While it can be used without engine groups or execution classes, it often makes sense to first use engine groups or execution classes to isolate applications from each other as aborting a query batch should be a last ditch effort and really aimed at applications that use adhoc queries or provide parameters that can be too wide (such as date ranges). The reason for this is that if a normal user application’s query aborts for any reason, the following is likely to happen:

- The user is simply likely to retry the query by pressing the ‘submit’ or ‘run’ button again
- If it is a batch that includes DML, valid business transactions can be lost by preventing the batch from running to completion.

Consider the following use cases for particular resource limits:

Limit Type	Usability	Use Cases
row_count	medium	• Prevent Cartesian Products • Abort ad-hoc queries with too broad of a parameter range for the type of application
elapsed_time	high	• Block bad queries from running infinitely • Resolve blocking by aborting the blocking or blocked query batches • Prevent CPU or IO hogging
io_cost	low	• Block bad queries from running infinitely (alternate method)
tempdb_space	high	• Prevent Cartesian Products • Prevent tempdb log suspends

Sometimes resource limits are imposed temporarily to stabilize a system. For example, if tempdb suddenly is filling frequently, temporarily adding a resource limit for all applications on tempdb_space is a valid option until the culprit can be identified. In such cases, just about any mix of resource limits can be used. Under normal circumstances, however, it becomes more of an application by application basis as some of the resource limits may not make sense. For example, consider IO cost. As a costing consideration, the value is determined by the IO cost computation of (25*Physical IOs+2*Logical IOs). One problem that should be noted right away that in today's larger memory systems, the io_cost resource limit will be harder to enforce as at some times less physical IO's may be required than others. In addition, anyone who has looked at the optimizer estimates well knows that the estimates in this area can be astoundingly high – and that while this applies to the final plan, even then the estimate can be off significantly. So the reality is that this really ends up more of a restriction on physical IO's during actual query execution - which at first glance suggests helping to prevent the IO subsystem from being overwhelmed. However, what limit do you impose?? And if you impose the limit, what prevents the user from repeatedly re-issuing the query, inflicting the exact IO subsystem saturation you were trying to prevent?

The row_count limit is similar. First, it only is enforced on the number of rows returned. An undesirable query using aggregates is likely to return fewer rows than normal queries, for example. Secondly, it is measured as the output of the query results vs. rows actually sent to the client (think select/into #temp. The result is for day-to-day operations, any resource limit on row_count has to be set fairly high and likely only for applications that support ad-hoc queries – whether the user is entering the query text directly or using query-by-example input parameters. This of course would raise the question of whether using the resource limit is a work-around to the application supporting a row count limit or as an alternative to setting the rowcount at the session level due to the degree of impact it has.

This is the reason for the usability column in the above table. It points out which resource limits make more sense during normal day-to-day operations. Consider the following table of suggested resource limits and when to employ them:

Limit Type	Application Type	Employ	Limit	Rationale/Notes
elapsed_time	Ad-hoc query tool	Normal	(time range dependent)	Block bad queries from running infinitely Prevent CPU or IO hogging
tempdb_space	OLTP	Normal	20MB	Prevent tempdb log suspends
tempdb_space	Reports/Batch	Normal	100MB	Prevent tempdb log suspends
row_count	As necessary	Stabilization	See notes	Tempdb should already be protected from impact of Cartesians, so this limit is to prevent the rows returning to the client from crashing the client application due to running out of memory (assuming it can handle the resource limit error without fatal exception).
io_cost	As necessary	Stabilization	See notes	Use QP Metrics to identify queries and limits by filtering on logical IOs. Then use io_cost resource limit to block queries from running that are causing excessive problems.
elapsed_time	OLTP	Stabilization	1-2 minutes	User gets an exception within 1-2 minutes instead of an indefinite hang causing the application to abort when user kills it.

As implied in the above table, the best use of resource limits is to protect system availability by aborting queries that are likely to consume excessive resources that can endanger all the users of the system. A second use case is to prevent run-away queries from reducing system performance. In a sense, then, resource limits are more of a 'protector' to stop problems from endangering the system whereas engine groups and execution classes are more 'containerization' utilities to provide predictable service levels.

Time Ranges

Unfortunately, time ranges as a resource governor in ASE only apply to resource limits vs. execution classes or engine groups. As a consequence, the most effective use of time ranges is to restrict users from running resource intensive queries during the wrong timeframes. For example, a common situation in business environments is that there is a mix of applications and users. Generally speaking, the normal OLTP and batch process applications should have well defined performance envelopes and should not normally need resource limits to control them under normal circumstances. However, in many companies, business analysts have direct access to the OLTP systems and often use query tools to submit adhoc SQL queries to the system. While an engine group, separate tempdb, etc. can help isolate some of the impact from other users, such analytical queries often result in large amounts of IO being submitted to the IO subsystem, which impacts overall system performance. This may be particularly undesirable during known periods of peak activity such as early morning, post-lunch or pre-market close rush. As a result, the DBA may wish to define time ranges during which the business analyst queries that exceed a specific io_cost or elapsed_time are simply aborted for the analyst to run at a later non-peak period.

To define a time range, the following commands are used:

```
-- sp_add_time_range <name>, <startday>, <endday>, <starttime>, <endtime>

exec sp_add_time_range 'morning_peak', 'Monday', 'Friday', '09:00', '10:00'
go
```

Some might find the syntax confusing and wonder if this defines a single time range that is all inclusive. It doesn't. For example, the above syntax does not mean a single time range starting Monday morning at 9am and ending Friday morning at 10am, but rather as expected, it defines a time range from 9am-10am on Monday through Friday of the week. However, since a resource limit can only specify a single time range, in order to enforce our earlier limits on nasty business analyst queries, we would have to not only define the time ranges, but individually bind the resource limits for each time range as follows:

```
-- first define the time ranges
exec sp_add_time_range 'morning_peak', 'Monday', 'Friday', '09:00', '10:00'
exec sp_add_time_range 'post_lunch_peak', 'Monday', 'Friday', '13:00', '14:00'
exec sp_add_time_range 'afternoon_peak', 'Monday', 'Friday', '15:30', '16:30'
go

-- now restrict the analysts…if there are a lot, we may wish to restrict the application
-- they use instead of loginname…in this case, we will limit them to queries that run
-- longer than 10 seconds
exec sp_add_resource_limit null, 'dbisql', 'morning_peak', 'elapsed_time', 10, 2, 2, 2
exec sp_add_resource_limit null, 'dbisql', 'post_lunch_peak', 'elapsed_time', 10, 2, 2, 2
exec sp_add_resource_limit null, 'dbisql', 'afternoon_peak', 'elapsed_time', 10, 2, 2, 2
go
```

An important step of this process is to notify the users ahead of time that they will have limits placed on them. If not, the unfortunate affect of the above is that some users might simply keep re-issuing the query – and while the resource limit might abort it each time –the

aggregate effect might be worse than the query running to completion. Additionally, if a user does hit a limit and an error is raised, the DBA may want to call that user to remind them of the limit and offer to help with the query to see if the query was simply bad SQL and can be rewritten in such a way that it runs within the defined limits.

Enabling Virtualization: Transportability

One of the side effects of the enormous jump in processor capacity in recent years is the renewed interest in virtualization – specifically using virtual machines – to consolidate multiple workloads in a larger host environment. Within the virtual machine software packages, some even support workload management concepts that support relocating executing processes from one host to another. One of the problems with this is that it involves relocating the process vs. relocating the application workload. For example, if 10 applications are all using the same ASE server, the virtual machine containing the ASE server can be relocated – but the applications all go with it. While this does provide the ability to deconflict the workloads between different processes sharing the same host but in different virtual machines, it doesn't help if the ASE server is simply reaching capacity due to sustained business growth, new applications, etc. The problem is that using a virtualized environment as the level of resource management, there is no notion of moving applications from one ASE server in one host to a different ASE server. Consequently, *in addition to* using virtual environments, it is important that DBA's configure the ASE servers to support the notion of transportable workloads, so that as business grows, the workload can be easily *divided* among multiple machines. This is especially key in shared disk cluster situations as the workload can be automatically load balanced and can result in unpredictable behavior if the servers were not set up to support transportable workloads.

Services vs. Servers

Traditionally, Sybase DBA's have thought of the interfaces (sql.ini) file entries in terms of a *server* name. In fact, the entry is a *service* name. The intent was to provide a logical mapping at clients to a physical resource masking the physical location. Consequently, a single service entry in the interfaces fill could have multiple physical entries associated with it – commonly used by DBA's for failover availability such as:

```
SYPROD
        query    tcp      ether    bigmachine1.domain.com    30000
        query    tcp      ether    bigmachine2.domain.com    30000
```

As we are well aware, ASE attempts the connection first on the first query address and then if that fails attempts to connect using the second query address. However, assume we have several main applications – clinical labs, triage, pharmacy and billing – all co-hosted on the same ASE server 'SYPROD'. In most customer locations, the interfaces file would resemble the above – a single entry to the single server. However, consider the better alternative:

```
CLINICAL
        query    tcp      ether    bigmachine1.domain.com    30000
        query    tcp      ether    bigmachine2.domain.com    30000

TRIAGE
        query    tcp      ether    bigmachine1.domain.com    30000
        query    tcp      ether    bigmachine2.domain.com    30000

PHARMACY
        query    tcp      ether    bigmachine1.domain.com    30000
        query    tcp      ether    bigmachine2.domain.com    30000

BILLING
        query    tcp      ether    bigmachine1.domain.com    30000
        query    tcp      ether    bigmachine2.domain.com    30000
```

All four entries point to the same server. There is a major key difference. In the first case, if the ASE was running in a virtual machine, however, and capacity became an issue, the VM would have be expanded – which might not work due to hardware restrictions. The only option left would be to move the entire VM to a larger host.

In the second case – service names – the system could be split and some of the processing moved to another ASE instance in a different VM on a different host. For example, if the huge batch process of generating patient bills started impacting the performance of patient triage, the BILLING service with its entire database(s) could be moved to a different machine or VM. Some might have been quick to note that this might not be so fast given the need to fix the interfaces file with the new machine name, but it could be made extremely easy if instead of a physical hostname, the services listed logical hostnames such as:

```
CLINICAL
        query   tcp     ether   clinical_pri.domain.com   30000
        query   tcp     ether   clinical_alt.domain.com   30000

TRIAGE
        query   tcp     ether   triage_pri.domain.com     30000
        query   tcp     ether   triage_alt.domain.com     30000

PHARMACY
        query   tcp     ether   pharmacy_pri.domain.com   30000
        query   tcp     ether   pharmacy_alt.domain.com   30000

BILLING
        query   tcp     ether   billing_pri.domain.com    30000
        query   tcp     ether   billing_alt.domain.com    30000
```

The above notion of defining services as logical entities on top of physical server instances is slightly more formalized in the notion of "Logical Clusters" in ASE Cluster Edition. One of the advantages of ASE/CE in that context is that when workload needs to be relocated, it is more transparent to the end-user as the connections are migrated when idle. Manual service entries as listed above would require an outage for the applications supported by the databases as the databases would need to be unmounted (see discussion on transportable databases below).

Transportable Databases

The concept of transportable databases was introduced in ASE 12.5 with the unmount/mount database commands. A key restriction is that databases are only transportable between servers using the same pagesize. The syntax is:

```
unmount database <dbname_list> to <manifest_file> [with override]
```

An important aspect of this is that while you are listing databases in the command, you are actually working with the set of devices that support those databases. Consequently, if you have more than one database sharing the same device, attempting to unmount one of them without unmounting the others will result in an error similar to:

```
unmount database pubs2_xml to 'c:\work\ASE_Tests\pubs2_xml.manifest.mft'
go
Could not execute statement.
UNMOUNT DATABASE: Device 'datadev01' is also used by database 'pubs2_a',
which is not specified in this command.
Sybase error code=14538
Severity Level=16, State=1, Transaction State=1
Line 1
```

The optional 'with override' clause is not meant to circumvent that problem – instead it is meant to allow databases to be unmounted that are isolated device-wise – but use encryption keys or declarative references to other existing databases not being unmounted. Remember,

ASE physically unmounts the devices and ceases to use them, so attempting to force an unmount on databases that are not isolated device wise would leave the database not being unmounted but sharing those devices in a suspect state.

If the database is to be physically relocated, the physical devices and the manifest file all need to be transported to the new host. In the case of a SAN-based ASE instance using raw partitions, this is often as simply as unmounting the devices from the host, changing the host to export them to in the SAN, then mounting them on the new host and finally ftp'ing (in binary mode) the manifest file to the new host machine. It gets a little bit more time consuming but not much more complicated when using file system devices – once the database is unmounted, you can opt to clone the disks using SAN utilities or opt to use the much longer process of rcp'ing/sftp'ing the files to the new host. Unfortunately, that's the rub of this process. The easiest way to ensure that a single database is using isolated devices is to use file system devices and DIRECTIO. However, that makes it just slightly more time consuming – unless those files are on dedicated file systems – which would have the same issues as raw partitions with respect to space optimization (with respect to device allocation not any inefficiency of how ASE manages space). However, this is where SAN-based systems excel as SAN systems can expose multiple slices of physical disks as fairly granular disk LUNs or Raidsets to be mounted. Additionally, the time for the file movement could be hidden by starting the disk cloning far in advance of when necessary and at the appointed time, make sure the disk cloning is in sync, perform the unmount, check to make sure still in sync and then simply split the disk pair(s).

The mount command is a bit more complicated. In its simplest form, the syntax is:

```
mount database <dbname_list> from <manifest_file>
```

This works very simply – ASE reads the manifest file to get the list of databases and devices, and then opens those devices and mounts the databases. However, the databases are still offline. For example:

```
mount database demo_db from 'c:\work\ASE_Tests\demo_db.manifest.mft'
go
Started estimating recovery log boundaries for database 'demo_db'.
Database 'demo_db', checkpoint=(134427, 18), first=(134427, 18), last=(134427, 18).
Completed estimating recovery log boundaries for database 'demo_db'.
Started ANALYSIS pass for database 'demo_db'.
Completed ANALYSIS pass for database 'demo_db'.
Started REDO pass for database 'demo_db'. The total number of log records to process is 1.
Completed REDO pass for database 'demo_db'.
Recovery of database 'demo_db' will undo incomplete nested top actions.
Started recovery checkpoint for database 'demo_db'.
Completed recovery checkpoint for database 'demo_db'.
Started filling free space info for database 'demo_db'.
Completed filling free space info for database 'demo_db'.
Started cleaning up the default data cache for database 'demo_db'.
Completed cleaning up the default data cache for database 'demo_db'.
MOUNT DATABASE: Completed recovery of mounted database 'demo_db'.
Execution time: 9.86 seconds
go
use demo_db
go
Could not execute statement.
Database 'demo_db' is currently offline. Please wait and try your
command again later.
Sybase error code=950
Severity Level=14, State=1, Transaction State=1
Line 1
```

The database is brought back to the same point as if you just completed a load database command. Consequently, you need to issue a 'online database' command to bring it back online. This sounds like an unnecessary step, but actually it provides some interesting flexibility

with respect to backups. For example, you can do a physical backup of the database by using the command:

```
quiesce database <tag_name> hold database_list for external dump to <manifest_file>
```

Then take the normal subsequent transaction log dumps. You can later mount the physical dump and since it is not online, restore the subsequent transaction log dumps to fully bring it up to date.

Unfortunately, unless you have established a global device path naming standard that is unique or similar as far as database names, you may be facing a slight challenge in that the device paths are different on the new host or a database name already exists with that name or a logical device (sysdevices.name) already exists with that name. Hence the syntax supports an in-line method to rename the database or redirect the mount point on the fly.

```
mount database all | database_mapping[, database_mapping, ...]
        from "manifest_file"
        [using device_mapping [, device_mapping...]
                [with listonly]

        database_mapping:
                        origdbname as newdbname
                |       newdbname = origdbname
                |       origdbname
                |       newdbname

        device_mapping
                        logical_device_name as new_physical_name
                |       new_physical_name = logical_device_name
                |       original_physical_name
                |       new_physical_name
```

The best technique to start with is to issue the mount command with the *listonly* option to first see the list of databases and devices involved:

```
mount database all from "/opt/sybase/testdb_manifest.mfst" with listonly
go
```

Remember, databases allocate space using *logical* device names – so one of two problems could exist:

- The physical path to the device differs due to a different mount point, but the logical device name is fine
- There already is a logical device with the same device name
- Both.

The first one is quite easily fixed by using the [new_physical_name = logical_device_name] mapping such as the following example:

```
mount database all from "/opt/sybase/testdb_manifest.mfst" using
        "/opt/sybase/syb15/data/GALAXY/test_data.dat" = "test_data",
        "/opt/sybase/syb15/data/GALAXY/test_log.dat" = "test_log"
```

If the second case (logical device name already exists) – ASE will fix this automatically by generating a new name for the device. So, essentially, the above command would work in all three cases assuming the physical device path was correct.

As the documentation notes, mounting a database does several things:

- Replication is turned off.
- Audit settings are cleared and turned off.
- Component Integration Services options, default remote location, and type are cleared.

- Cache bindings are dropped for both the mounted databases and their objects.
- Recovery order is dropped for the mounted databases and becomes the default dbid order.

Consequently, prior to unmounting a database, you should at least extract:

- Any cache bindings
- Proxy table remote locations (especially if the default location was used)

These will have to be re-instated at the new location.

While mount/unmount/quiesce database has many uses, the principal focus for this discussion is on enabling DBA's to split the load from one server or VM when it reaches resource saturation and then spread the load across multiple new systems with as minimal disruption as possible. Consider our earlier example in which possibly a previous consolidation ended up with several large applications sharing the same host.

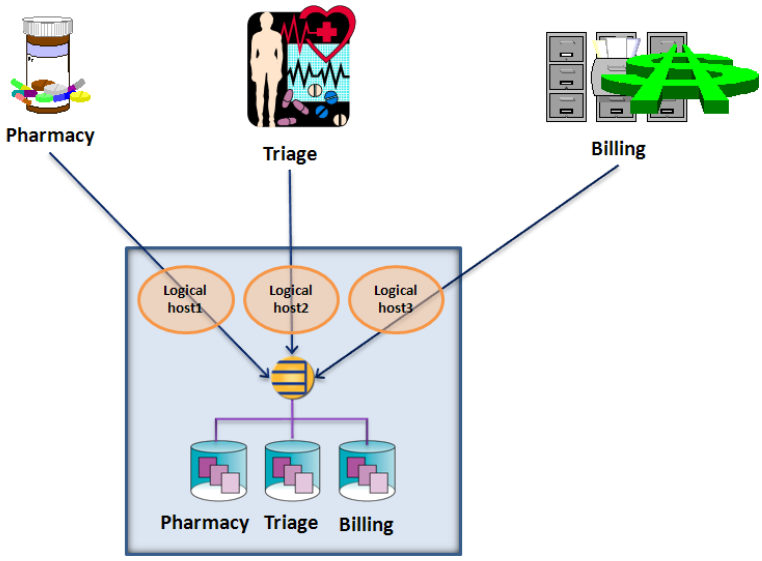


Figure 22 – Example Consolidated System

As business grows, resource contention starts. As noted above, as long as each database used isolated devices and logical hostnames were used and application service names in the interfaces file, the workload could quite easily be redistributed in minutes (if scripted) to:

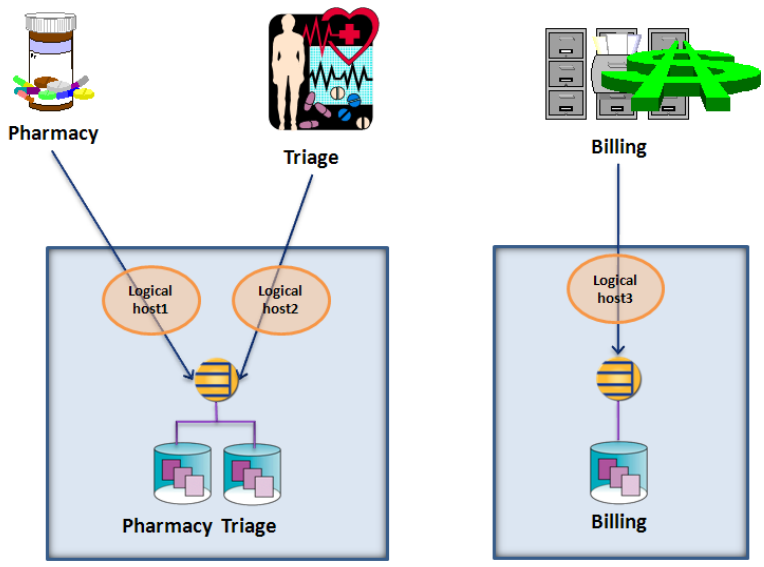


Figure 23 – Re-Balancing Workload via Transportable Databases

In both cases, the existing databases and applications on each server would not be impacted other than those being transported. The sequence would be similar to:

1. Synchronize logins and roles (can be done in advance)
2. Use the dynamic listener feature of ASE to stop the ASE from listening on logical host3
3. Unmount the "Billing" database
4. Relocate the devices using SAN facilities
5. Mount the "Billing" database on the second physical host
6. Start the logical host3 on the second physical host
7. Fix cache bindings and proxy remote host mappings
8. Use the dynamic listener feature to start the listener on logical host3

The only application impacted during this would be the "Billing" application. Obviously, this could be done occasionally – but not too often or the impact on the transported application may become an issue. If frequent workload redistribution is required, ASE/CE is a better choice as it can dynamically and transparently rebalance workload using logical clusters.

Named Caches & Tempdbs

In any workload re-distribution effort, the effort is made easier if the application's named cache and tempdb requirements leverage generic caches/tempdb's aligned according to the cache or tempdb profile needed vs. specific application tempdbs or named caches. For example, earlier, we discussed that the following named caches were likely necessary:

- Log cache (1-3) – for the transaction logs
- System table (1) – for the system tables
- tempdb caches (1 per tempdb or tempdb group)
- Restriction (1 or more) – for application logging tables or BLOB data columns

- Reference (1) – for small lookup tables
- Hot Static Tables (1+) – static size/fixed length field frequently updated tables that can be fully cache
- Hot Volatile Tables (1+) – more volatile tables or their indexes that are frequently updated or inserted
- Application Specific – as necessary

The last one is a bit of a problem in setting up for transportable databases. No one would argue the fact that in consolidated systems, sometimes it is necessary to separate the data caching requirements for different applications. Fortunately, dropping a cache is dynamic – as is creating a new cache. But you need to be careful with what you do with the regained memory from dropping an application cache. If you add it to an existing cache, any later rebalancing efforts may run into a problem as resizing a cache is not dynamic. This also could affect the transported database in the new system as existing applications may have already fully divided up the available memory and assigned application specific caches…and now there is a new requirement. While the new requirement could be possibly carved out of default data cache (if large enough), there could be impact on the existing applications that leveraged the default data cache in addition to the application specific caches.

Tempdb faces a similar issue. Earlier it was stated that applications should likely have their own tempdb or tempdb group. One of the more problematic issues is that you cannot unmount user created tempdb's – which means when a database is moved from one system to another, the best you could do is the following:

- drop the temporary database at the original server
- drop the devices from the original server
- relocate the devices to the new system (using SAN facilities if SAN disks)
- add the devices to the new system
- recreate the application specific tempdb (this could take awhile)

The third step is a bit of a problem. If using SSD's that are internal (such as backplane based PCIe SSD's or internal SAS/SATA2 drives), moving them is difficult and would require both machines to be shutdown. If using an in-memory tempdb, we arrive back at the problem of pre-existing cache configuration in the new server if already hosting applications. If using cached file system devices, you either need to move the file system (and hopefully nothing else was on it) or you have to have the space available on a file system on the new system. In one sense, if the space is available, this is probably the easiest to deal with as you can allocate the space and re-create the tempdb(s) prior to actually moving the database itself.

The point is that in both cases, while application specific tempdbs or named caches may be necessary, it should be approached cautiously to be sure that it really is necessary vs. leveraging the generic caches or tempdbs pre-created according to the recommendations. If they are used, it may point to a need to plan the application move more in advance and approach it in phases to allow the downtime necessary to reallocate hardware, resize caches, etc.

Logins & Roles

Fortunately, logins and roles are far easier to deal with as both are simply data rows in ASE. Most enterprise customers – especially those who have worked with Replication Server previously – are well versed in the steps it takes to synchronize logins and roles across different servers.

In larger environments, it might be thought that it would be much easier to simply use LDAP or Kerberos for authentication – which avoids the problem posed by systems at different ASE versions and changes in password encryption that have happened recently – changes which complicate synchronizing logins. But as those that have done this can tell you, the more fun challenge of this process is aligning the SUID and SRID (server role id from syssrvroles) – not only in the master database – but in the sysusers table within each database.

Many customers that have Replication Server or have large ASE distributions simply have adopted techniques that ensure that when a login is added to a server that it uses a pre-assigned SUID (such as the employee id) that avoids the problem. If it is expected (due to projected growth) that you may be transporting databases frequently, it is highly recommended that you adopt a similar scheme – and extend it to include ASE roles.

However, if you find you need to do it manually, the current steps for ASE 15.x is:

- bcp out syslogins from original server
- extract the ddl for any roles
- Extract the 'EL' rows from sysattributes (external logins)
- create a copy of the syslogins, sysloginroles and syssrvroles tables in the new system as user table in the master database
- use select/into to create copies syslogins, sysloginroles and sysroles tables (for recovery purposes) (for reference, call them syslogins_recovery, sysloginroles_recovery, sysroles_recovery to avoid confusion)
- bcp in syslogins, sysloginroles and syssrvroles to the user tables
- create the roles using the DDL (note that when a role is created, it is automatically added to the sysroles table in each database
- sp_configure 'allow updates', 1
- find logins that exist in the old system but not in the new and check for suid conflicts
- Use insert/select to insert the new logins – adjusting suid as necessary for the conflicts
- Insert any missing roles for logins by joining the user tables and new tables to create mappings using SQL similar to:

```
insert into sysloginroles (suid, srid, status)
select nl.suid, nr.srid, olr.status
  from master..syslogins nl,
       master..syssrvroles nr,
       master..old_syslogins ol,
       master..old_sysloginroles olr,
       master..old_syssrvroles or
  where nl.name = ol.name
    and ol.suid = olr.suid
    and olr.srid = or.srid
```

```
        and or.name = nr.name
```

- Insert the extracted 'EL' rows into sysattributes

If you make a mistake, the recovery process is as follows:

- clear sysloginroles back to the default

```
delete master..sysloginroles where suid>2
delete master..sysloginroles where srid>15
```

- clear syssrvroles back to the default

```
delete master..syssrvroles where srid>15
```

- clear syslogins back to the default

```
delete master..syslogins where suid > 2
```

- insert from the syslogins_recovery table all but the sa and probe users

```
insert into master..syslogins
select * from master..syslogins_recovery where suid>2
```

- re-insert the non-system roles from the syssrvroles_recovery table

```
insert into master..syssrvroles
select * from master..syssrvroles_recovery where srid>15
```

- re-insert the sysloginroles_recovery data

```
insert into master..sysloginroles
select * from master..sysloginroles_recovery
  where (suid=1 and srid > 15)
     or (suid>2)
```

Assuming everything is fine, then, after the transported database is brought online:

- update the sysusers table using a query similar to:

```
update sysusers
   set suid=l.suid
   from syslogins l, sysusers u
   where u.name = l.name
```

- resync the sysroles table

```
truncate table sysroles
go
insert into sysroles select * from master..sysroles
go
update sysusers
   set gid=r.lrid,
       uid=r.lrid
   from sysusers u, sysroles r, master..syssrvroles s
   where u.name = s.name
     and s.srid = r.id
go
```

Note that each database should use the same recovery technique used for master – only focusing on sysusers, sysroles – being careful to avoid removing dbo (uid=1), public (uid=0), guest (uid=2) and the system groups (uid between 16384 and 16397).

Obviously, this is a process that should be tested and verified each time using a test server prior to doing in a production system. But the risk involved strongly advocates using a standard scheme to avoid having to use it in the first place.

ASE Cluster Edition & Cluster Scalability

In 2008, Sybase began introduced ASE Cluster Edition (ASE/CE) - its third generation of clustering following Sybase MPP (nee Navigation Server) and Sybase ASE/HA Companion Servers. It is important to realize that the focus of each of these releases were considerably different. First, ASE/HA Companion Servers were introduced for VAX/VMS systems in the 4.x timeframe and then later for mainstream Unix OS's for the 12.0 and 12.5 releases. Companion Servers as discussed earlier in the hardware discussion focused strictly on high availability by minimizing the downtime due to single node failure. The second cluster implementation released by Sybase was Sybase MPP – an Massively Parallel Processing implementation focused on decision support systems. ASE/CE was designed primarily as an availability solution that did not suffer from the scalability limitations imposed by Companion Servers for shared data access and also provided better hardware resource utilization for consolidated systems. It is important to recognize that first and foremost, ASE/CE is an ***availability*** solution. This is not unique to Sybase ASE/CE as nearly all true pure shared disk clusters have the same focus – no matter what the marketing claims might be.

The good news is that some degree of scalability can be achieved. While not nearly as scalable as a single large SMP host, several smaller SMP hosts in a SDC cluster can achieve decent scalability if the database administrators and application developers work closely together at architecting the solution and each understands how the technology works to best define the cluster implementation. The white paper *"Understanding Availability & Scalability with ASE 15.0 Cluster Edition"* describes in detail the availability features of ASE/CE along with the scalability concerns. This paper borrows heavily from that paper with respect to scalability but makes recommendations based on the current capabilities of ASE/CE vs. treating it from a strict academic standpoint.

However, if moving an application from a single ASE instance into a cluster, the following considerations should be factored into the hardware sizing and configuration:

- Cluster overhead for distributed locking, etc. in ASE/CE is about 10GB in 16-20GB memory configurations. As a result, cluster hardware nodes should have a minimum of 64GB of memory and have at least 10GB more allocated to ASE/CE than the single instance.
- Cluster overhead for CPU usage is about one full engine per instance. Plan on at least one engine and one core per instance per hardware node.
- A 10Gb Ethernet card is cheaper and faster than network bonding multiple 1Gb NICs'. Use 2 10Gb Ethernet cards for the private interconnects along with 2 10Gb switches.
- The current sweet-spot for application consolidation is 3-4 instances/nodes. If a single application is being moved to a cluster, unless the application can be partitioned and the partitioning done correctly, it will be difficult to scale beyond one node – implying ASE/CE will be primarily a HA configuration.

The rest of this section will detail some of the other aspects to cluster scalability

SDC Basics: Cache Synchronization, Distributed Locking & IO Fencing

In order to achieve a single system image, any shared-disk cluster DBMS has to implement three critical components:

- Cache Synchronization
- Distributed Lock Management
- IO Fencing

These three components plus others provide the DBA with a much easier to manage system that other clustering solutions such as shared-nothing MPP clusters or even HA/Companion Server clusters. In fact, by comparison, a shared-disk cluster could be characterized as:

- Low complexity for development and administration (single system view)
- High complexity for DBMS Kernel (buffer mgmt & locking semantics)
- Low complexity for DBMS Query Processor/Optimizer
- Smaller/more frequently (high volume) interconnect requests

The reasons why are evident in the following sections on the cache synchronization, distributed lock management and I/O fencing. The question is - with all this complexity and difficulty in building a shared-disk cluster, why would a DBMS vendor choose this strategy over a shared-nothing approach. The answers are that shared-disk clusters:

- Support unpartitioned applications better than the shared-nothing approach which requires a strict partitioning scheme.
- Are better able to adapt to changing workloads - especially in the OLTP and mixed-workload environments whereas shared-nothing is largely a DSS implementation.

Cache Synchronization

Shared-disk clusters seek to resolve the manageability of clusters while providing some scalability through a different technique for scalability and single system image – ***cache coherency***. In cache coherency (aka cache fusion) implementations, instead of shipping query fragments to where the data resides, the raw data pages are instead shipped to where the query is executing using cache synchronization. However, each instance still has its own separate memory that is not shared in which copies of the same data pages could reside. Note that the most common technique used to ensure cache coherency in SDC implementations to date is cache synchronization – shipping the data from one data cache to the other. An alternative implementation is disk synchronization in which requested pages are flushed to disk and the requester does a physical read from disk. Early Oracle 8i implementations used this technique for example. However, as network speeds started overcoming disk IO speeds, this technique has been almost universally replaced with cache synchronization.

Cache coherency and cache synchronization in general poses several problems.

How do you know if someone else has a data page in memory or what state it is in (e.g. dirty)?

How do you know if you have the most current up-to-date copy of a page?

How do you avoid multiple users on different instances attempting to change the same data simultaneously?

What happens if one instance gets separated (network-wise) from the rest?

Much like distributed query processing is the back-bone for shared-nothing clusters, cache coherency is the central capability that shared-disk clusters are based on. In a cache coherency system, the cluster-ware tracks which pages were most recently modified by different nodes and ensures that requests for data pages are satisfied by supplying the most current copy of the data page - whether from disk or by transferring from the memory of another instance. Consider the following sequence of states for a simple operation:

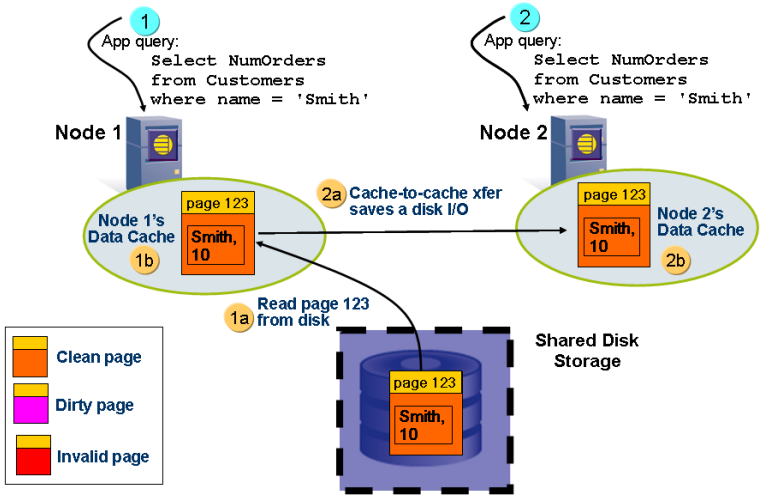


Figure 24 - Cache Coherency: Subsequent Reads

In the above diagram, two different users ask for the same row of data - one after the other. The first query (1) retrieves the data page containing the row (1a) and then populates its data cache (1b). Then the second users query executes (2). At this point it should be obvious that some form of *directory service* is necessary so that the second node is aware that the page is already in node 1's cache and can simply do a cache synchronization operation (2a) to populate its data cache (2b). In reality, the directory service would have been required even for the first query to determine that the page was not already in cache within the cluster and that a physical read was required.

What happens if the page is updated? Consider the following illustration:

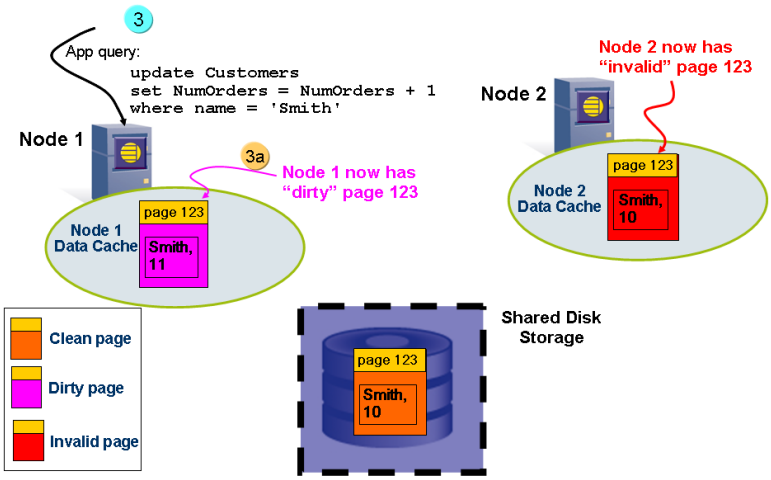


Figure 25 - Cache Coherency: Data Modification

At this stage (3) as depicted, Node 1 has a more up-to-date copy of the page reflecting the most current order while the page in Node 2 has become "stale" and invalid. Now, assume Node 2 attempts to requery the same page. Technically, it is in cache (although invalid) but it has no way of knowing this unless each write operation updates all the caches at a tremendous overhead cost. Instead, for each query/page access, the instance determines from the directory service if the version of the page in its cache is the most current. If not, it simply requests a cache transfer from the location that has the most current page.

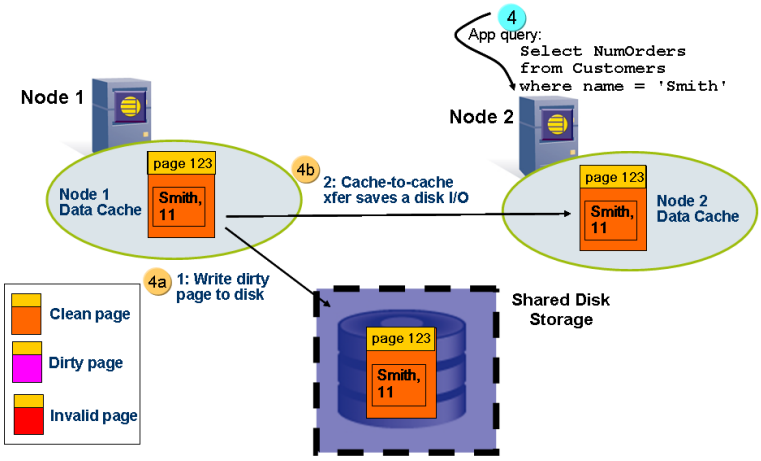


Figure 26 - Cache Coherency: Cache Synchronization

Note that in the above sequence, the dirty page flushed to disk *before* the cache transfer. The reason for this is to ensure that a timing issue does not lead to an incorrect page version on disk. For example, consider what would happen if Node 2 now also updated the page but that the page write (4a) had not been completed. In normal DBMS data caching strategies, the database log is implemented as a write-ahead log to ensure recoverability. The dirty data pages are then flushed asynchronously - e.g. for ASE the writes are initiated by the wash marker, the housekeeper or the checkpoint process. With two separate instances, these asynchronous events could occur at different times, which means that for the above scenario, it would be possible for Node 2's physical write to occur ahead of Node 1's. From a recovery perspective, this is not a problem as the log sequence would still roll forward the changes correctly. But what would happen if eventually the page was removed from cache by normal MRU/LRU

processing and then a subsequent query attempted to re-read the page? The unintended result would be that since Node 1's physical write occurred later, that it would be the version on disk and the effect would be as if Node 2's transaction had never occurred. Consequently, prior to any cache transfer, any dirty pages are first flushed to disk to ensure that the latest version of the page is always on disk.

Distributed Lock Management

No matter whether using version based concurrency to avoid locking on reading or whether using lock based concurrency, at some point, the appropriate data rows and pages will have to be locked. Within ASE, this is implemented using standard locks organized into two lock hash tables - one for table locks and the other for page/row locks. Before a lock can be granted, the lock hash tables have to be searched for conflicting locks. As a simplistic consideration, assume that the hash function is a mod(10) of the page number. Assume we have a user wishing an exclusive lock on page 15 of a particular table. Using the hash function, we arrive at a value of "5" - which means that we look at hash bucket "5" and linearly scan for any conflicting locks. An illustration of this could be depicted as:

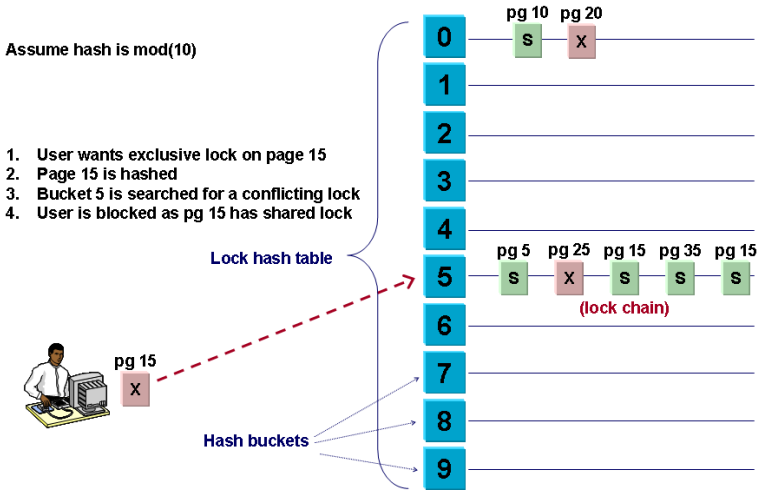


Figure 27 - Simplified Illustration of Lock Hash Table and Lock Chain Searching

In the above diagram, one of the obvious considerations is the length of the lock chain in each bucket. Since this is a linear search, the more locks - the longer the search and the more overhead of locking. This overhead is reported via sp_sysmon in the lock management section as illustrated below:

```
Lock Management
---------------

  Lock Summary                per sec      per xact       count  % of total
  --------------------------  -----------  -----------  ---------  ----------
  Total Lock Requests          434495.3         379.1   52139437        n/a
  Avg Lock Contention             347.0           0.3      41636        0.1 %
  Deadlock Percentage               0.0           0.0          0        0.0 %

  Lock Detail                 per sec      per xact       count  % of total
  --------------------------  -----------  -----------  ---------  ----------

  Table Lock Hashtable
    Lookups                    15177.9          13.2    1821353        n/a
    Avg Chain Length               n/a           n/a    2.11600        n/a
    Spinlock Contention            n/a           n/a        n/a        0.0 %
...
  Page & Row Lock HashTable
    Lookups                   266115.3         232.2   31933831        n/a
    Avg Chain Length               n/a           n/a    0.04418        n/a
    Spinlock Contention            n/a           n/a        n/a        0.0 %
```

Figure 28 - Sample Lock Chain and Spinlock Contention from sp_sysmon

As noted in the Sybase documentation, the 'lock hash table size' should be increased anytime the average lock chain length is greater than 5. In order to acquire or release a lock, the lock chain has to be modified. This entails grabbing the spinlock on the lock hash bucket, modifying the lock chain and then releasing the spinlock. By default in ASE, each lock hashtable spinlock controls 85 hash buckets. If the spinlock contention is high, the "lock spinlock ratio" can be tuned. Once the lock chain is modified, the process is notified that the lock is granted and it can modify the data without fear of concurrent access corruption. The same process is then followed to release a lock. Again, the concept of a lock chain is not unique to Sybase ASE - in Oracle the implementation is via the ITL which is associated with each row and has similar tuning considerations.

The point to all of this discussion is that the same process occurs in a cluster server as well - but at another order of magnitude as the processes acquiring locks may be in different servers. The question is:

> *Before acquiring a lock, should a process on one node first search all the other node's lock chains for a conflict? or….*
>
> *After acquiring a lock, should a process notify all the other nodes that it has the lock?*

Neither solution scales well. While at 2 or only a few nodes, the overhead of communicating with every node is not likely that great, as the number of nodes becomes 4 or higher, the overhead becomes increasingly debilitating.

The solution that both Sybase and Oracle use is to distribute the locks among the nodes and implement a lock directory service. Much like the lock hash buckets, the lock hash value is used to determine which node is the lock directory. Prior to acquiring a lock, a process first asks a *lock directory* which node is the lock master. The *lock master* is the node which has the lock chain for the specific page or row being requested. Once the lock directory reports which node is the lock master, the process then requests that the lock master grant it the lock. The advantage to this implementation is that the cluster locking overhead remains the same no matter how many nodes are in the cluster. The two - the lock directory and the lock master - together can be thought of as the directory service mentioned earlier that tracks which pages are in cache at any given point in time.

The lock master not only tracks which processes have locks on a particular page, but also which node was the last node to request an exclusive lock indicating a change on the page. The rationale is that a change to the page has been made and that change is likely still in cache at that node - consequently the last node to have an exclusive lock on a page or row is considered the *lock owner*. In order for the requesting node to get the latest copy of the page for consistency purposes, a cache transfer of the page has to be done from the last node to have an exclusive lock to the requesting node. An illustration of a non-conflicting lock transfer is illustrated below:

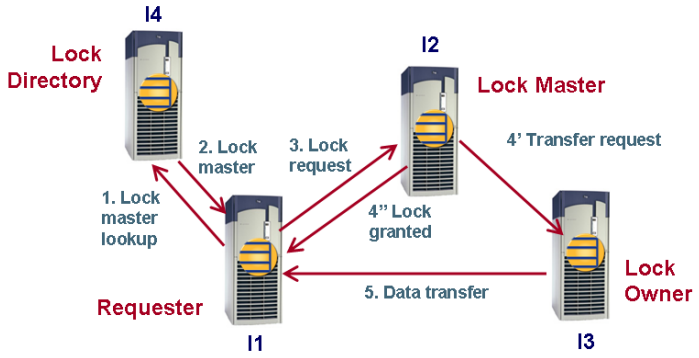


Figure 29 - Non-Conflicting Cluster Locking & Cache Synchronization

If there is a conflicting lock, the lock master tells the lock owner to downgrade its lock - which of course it cannot do until its transaction is finished. Once the conflicting transaction is finished on the lock owner, the lock owner notifies the lock master that it has downgraded its lock and does a cache transfer to the requesting node. This is illustrated in the following diagram:

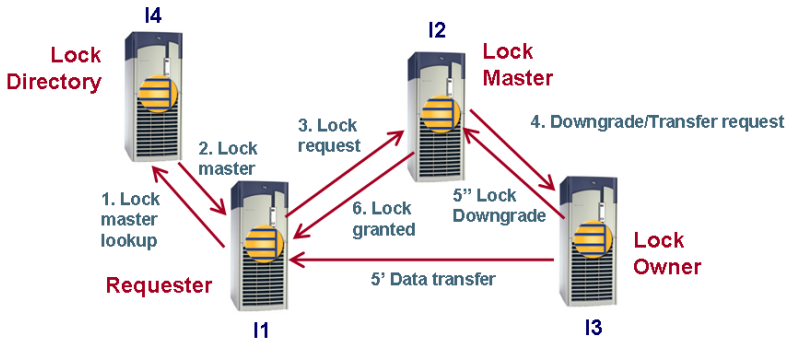


Figure 30 - Conflicting Lock Request & Cache Synchronization

Note that since ASE stores data pages (vs. rows), the unit of data transfer/cache synchronization is the data page. An addition note is that to ensure database recovery, dirty data pages are flushed to disk prior to any cache transfer. This helps speed recovery in the event of a crash as fewer transactions will need to be rolled forward - and also enables a disk based transfer alternative to network transfer.

Several observations should be obvious from these points:

- Locking a page or row will take slightly longer in ASE Cluster Edition.
- The more a data page is modified by different nodes, the higher the amount of cache synchronization requests and cluster overhead.

- The more cache transfers that occur for a single page, the higher the physical writes as possibly compared to non-clustered systems. This is due to the requirement to first flush a dirty page to disk before any cache transfers as discussed earlier.
- The more the data can be partitioned such that the requestor and owner nodes are the one and same, the fewer cache synchronizations need to take place and the less overhead (although the lock directory and lock master overhead may still exist)

This overhead results in numerous schema considerations as well as query issues. However, in ASE/CE, Sybase implemented several new types of locks – one of which facilitates scalability for read operations. As we all know, ASE uses a shared lock to prevent memory corruption while reading a page. If two different users on two different instances are reading the same data, the result could be a flood of distributed locking coordination between the instances. Consequently, in ASE/CE, once an instance releases the shared lock on a page, it retains a "retention lock" on the page. This adds a slight bit of overhead for writing applications as the lock manager needs to tell all the instances with retention locks to discard them. However, considering that reading is often the largest ratio in the read-write ratio, the tradeoff is more than beneficial.

IO Fencing

When multiple nodes connected over a network are all capable of writing to the same physical location, it is extremely easy to have a database corruption should one of the nodes lose contact with the others. This problem is often referred to as "split brain" in which one part of the cluster thinks the current cache state for a data page is one state while another part of the cluster assumes it is different. This could happen as follows:

- Two different nodes both read the same data page. Irrespective of the order of the read, both nodes currently have the page in cache.
- All the interconnects between those two nodes simultaneously experience failure.
- Both nodes get requests to modify the page from different users.
- Because the interconnect is down, each node assumes it is the survivor, re-masters the lock directory and believes that it as the sole survivor, it has full access
- Both nodes modify the page in memory and then flush the modified page to disk as would be normal.

The problem is that one of the nodes is wrong. When the other node attempts to read the page from disk later, it will hit a logical corruption as page pointers may be invalid or row locations may have moved, rows deleted or other considerations not reflected in the cache. IO fencing prevents this by allowing a coordinating process to register with the hardware controller a list of processes and hostid's that are allowed to write to the device . When the cluster controller determines that an instance of the cluster is no longer responding to heart beats, it simply removes the instance's information from the list registered with the hardware controller.

Sybase ASE/CE's implementation of IO fencing is implemented by using the SCSI-3 Persistent Group Reservation (PGR) facility in the SAN storage system. The best description of this implementation and how to set it up on various platforms can be found in the white paper *"ASE Cluster Edition: I/O Fencing Overview"*. The one point the paper stresses that is important from a hardware implementation viewpoint is that IO fencing is enforced at the LUN level as exposed by the SAN. Using OS utilities or volume managers to sub-partition that LUN into multiple devices could result in unexpected performance or stability issues.

SDC Basics: Logical Clusters & Workload Management

The above discussion on distributed locking, cache synchronization and IO fencing is all pretty much a common basic implementation by most shared disk clusters. It is important to understand those concepts as they are crucial for understanding the physical mechanics of what is happening within the cluster and then by understanding that, determine how to best segregate the workloads so as to minimize the impact. The management of these cluster mechanics is accomplished by layering an abstraction layer of cluster services that provide cluster resource assignments and workload management to optimize the cluster scalability by minimizing the need for distributed locking or cache synchronization. Sybase ASE/CE's implementation of this management layer contains two critical components: Logical Clusters and Workload Manager.

Logical Clusters

ASE Cluster Edition provides a distinctive feature called "Logical Clusters". A logical cluster is a single application's perspective of the entire cluster - which could be a subset of the available instances. Each application can have its own logical cluster; consequently, multiple combinations of cluster subsets can exist simultaneously. Each logical cluster has "*base*" instances in which normal processing occurs as well as "*failover*" instances which become available for the logical cluster if a failure occurs. Because a logical cluster can span multiple instances, administrators can manage the failover mode to be either "*instance*" or "*group*". In the case of "instance" mode, should any instance of the logical cluster fail, the workload immediately fails over to one of the failover instances. If an instance fails for a logical cluster using the "group" mode, the workload is redistributed within the surviving base instances unless no more base instances are available. The handling of failover and how availability is achieved in ASE/CE is outside the scope of this paper.

Consider the following diagram from an application consolidation/scalability perspective:

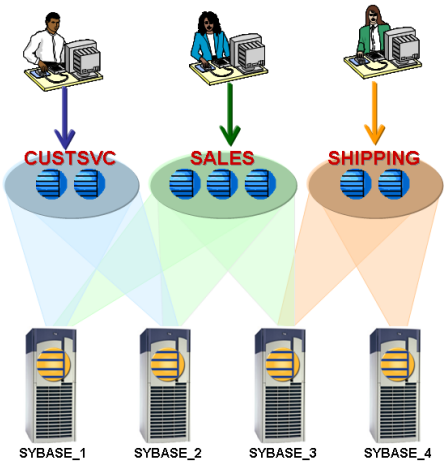


Figure 31 - Logical Clusters and Cluster Instances

The above diagram depicts three logical clusters: CUSTSVC (customer service), SALES, and SHIPPING - created on a single cluster with four instances: SYBASE_1 → SYBASE_4. In this depiction, the base instance associations are depicted via the colored shaded areas. When an application connects, *routing rules* are used to determine which instance(s) the application should be connected to based on the logical cluster associated with the application. If the application connects to the wrong instance, the connection is immediately redirected to one of the instances supporting the logical cluster. For instance, in the above diagram, if the SHIPPING user had initially connected to SYBASE_1, the cluster would automatically redirect the connection to SYBASE_3 or SYBASE_4. This is an important capability not only to separate the different applications sharing a cluster, but also to conserve "spare" resources for multiple mission critical applications.

It also is an important concept to reducing the impact of cluster overhead of distributed locking and cache synchronization. For example, in the above diagram, if the SHIPPING application/logical cluster had every node listed as a “base” resource, then application users and processing could happen on any node. As a result, it is much more likely that distributed locking and cache synchronization would be necessary – slowing down the individual transactions. By limiting it to two of the instances, the amount of distributed locking and cache synchronization is reduced significantly.

As illustrated above, logical clusters are a key feature if attempting to consolidate a number of different applications into a single cluster. While shared data is visible to each node, the applications can be much more easily separated when each application is assigned to its own logical cluster. A key concept to implementing logical clusters in ASE/CE is that few logical clusters should extend beyond more than one “base” node unless the other nodes are offline. If an application exceeds the capacity of a single node of the cluster, then it is likely that the application needs to be partitioned (later topic) and use multiple logical clusters with different instance assignments instead of trying to have the ASE CE workload manager make arbitrary decisions about how to balance the workload.

Applications & Logical Clusters

The first step is to clearly delineate the read/write relationships of the primary applications across the various databases. This is crucial for determining which logical clusters should be

aligned with the same nodes. The easiest way to do this is to simply list the major business functions for each application in a grid of the various databases. Consider the following example grid – note that the numbers are for example only.

Application	Module	Function	User	Volume /min	Trading Platform	CRM	Settlement	Market Data	System Audit	Staging	Historical Trading
Trade	Trade	Trade Order	Web	1800	SIU	S		S	S		
Trade	Analysis	Market Analysis	Web	3000				S			
BackOffice	Trade	Phone Trade	Internal	60	SIU	S		S	S		
Order Exec	OrderExec	Submit Order	Order exec	1800	U						
Audit	Audit Event	Trade Event	audit	1800					I		
BackOffice	CRM	BalanceReport	Internal	1		S					
Reuters	MarketFeed	Market Feed	(reuters feed)	100				I			

S=Select, I=Insert, U=Update, D=Delete

As you can tell, both the DBA's and the developers will need to work this out together. One of the most common mistakes is that DBA's & developers simply create one or two logical clusters or one per application and distribute them according to perception of load. This is a mistake – for example, the Order Execution application in the above has roughly the similar volume to the Trade application. The tendency would be to split the two to different logical clusters and nodes to try to balance the load. While the different logical clusters make sense, the problem is that the order execution writes to the same database within a very short timeframe (milliseconds) as the trade – and the multi-node writing would result in a performance degradation as a result of requiring a cache synchronization.

Generally speaking a typical application will use multiple logical clusters – depending on the module of the application. For example, if strictly using application to determine the logical clusters, the Trade and BackOffice applications in the above would have a single logical cluster each. The problem is that the load from the Trade application is heavily divided between two different functions – a single logical cluster would not help balance the load. Similarly, while it might be tempting to direct the BackOffice application to a separate node from external users, the BackOffice trades will result in multi-node writes and cache synchronization conflicts with the primary trade application.

Consequently, it is best to consider the different application modules and functions when defining the logical clusters and the node alignments.

Connection Pools/Application Connections

The impact on the connection pools or the connection string for non-pooled applications should be obvious. The application/module should connect to the appropriate logical cluster – and connect to the primary database that most of the write operations occur in. Consequently, the middle tier configuration will need to create a connection pool for each of the logical clusters the application server will access. A common mistake often seen today that contributes to SMP scalability problems (and use of engine groups) but especially with clusters is the typical middle tier configuration with very few ASE server based connection pools that are shared among a lot of applications.

Internal vs. External Users

One of the common considerations in creating logical clusters when supporting both internal and external users is whether to have a single logical cluster that supports both – or separate logical clusters. One reason for separate would be to allow one class of user to continue business processing in the event of hardware failure – as the different logical clusters could have different failover resources, modes (down-routing and fail-to-any attributes). For example, in the above sample applications and modules, both internal and external users are performing trades. A single logical cluster "Trace_LC" could support both classes of users. However, two logical clusters – "Trace_Web_LC" and "Trace_BckOff_LC" could allow BackOffice users to continue trading even in the event of multiple failures provided that the failover down-routing/fail-to-any attributes were set to support a higher level of application availability vs. the web users. Obviously not all logical clusters would need this sort of sub-division – but it is easier to configure at the onset than reconfiguring the system and application connection pools later.

Workload Manager

The workload manager in ASE/CE attempts to prevent a single instance of the cluster from becoming swamped – especially if other instances are much more idle. The workload manager provides both a manual or scheduled workload management capability as well as a dynamic workload management.

Logical Clusters & Scheduled Workloads

As mentioned earlier, ASE Cluster Edition provides support for application consolidation by providing workload management on a logical cluster basis. Of course, logical clusters provide some basic management capabilities inherently by being able to separate workloads to different nodes, etc. The problem is that workload resource requirements vary throughout the day and may need additional resources during normal peak processing periods. Consider the following simple example of a 2 instance cluster and example logical clusters for a mythical retail store:

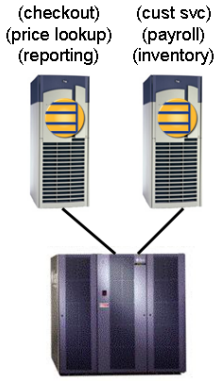


Figure 32 – Example 2 Node Cluster with Logical Clusters

For example, one possible workload distribution might be that on Monday mornings, all the other applications are relocated to the second instance while reports are run on the first.

Similarly, on Tuesdays and Thursdays when deliveries arrive at the store, the inventory management logical cluster may need priority.

Such requirements can be met using two aspects of ASE/CE. First, in addition to being able to be brought online and offline interactively, logical clusters also support the ability to have scheduled off-lining via the command:

```
sp_cluster 'logical', 'offline', <LC Name>, {cluster | <instance_name>},
                {wait | until | nowait}, <hh:mm:ss>, @handle
```

For example:

```
-- offline the reporting logical cluster on instance 1 at 4pm
declare @handle varchar(15)
sp_cluster 'logical', 'offline', 'Reporting', 'instance', 'instance2',
        'until','16:00', @handle output
print "Logical cluster will go offline at 4pm.  To modify, use handle: '%1!'",
        @handle
```

The only issue is that as a command line entry, this is a onetime event. This shortcoming is easily overcome using the ASE Job Scheduler, Unix cron, BMC's Control-M, CA's Autosys or other scheduling program.

Dynamic Workload Balancing

While logical clusters and the Job Scheduler can adapt for normal peak processing workload distribution, it may not help in balancing workload during unexpected periods - or when attempting to load balance across both instances for the 'Checkout' logical cluster in the example above. To solve this, ASE Cluster Edition introduces a new server level process called the "Workload Manager" that provides load distribution capabilities. The workload manager achieves workload distribution using two techniques:

- *Login Redirection* - in addition to logical cluster connections, login redirection is used to direct new connections to less loaded instances supporting the same logical cluster.
- *Dynamic Load Distribution (aka Connection Migration)* - to allow rebalancing of existing workloads, existing connections can be migrated to other less loaded instances for the same logical cluster.

In order to determine the relative workload, the workload manager collects the following metrics for each instance:

- *User connections* – the capacity of an instance to accept a new connection, based on resource availability (think sp_configure 'number of user connections).
- *CPU utilization* – the capacity of an instance to accept additional work. Essentially the engine utilization.
- *Run-queue length* – the number of runnable tasks on a system. Run-queue length measures the processing backlog, and is a good indicator of relative response time. A runnable task is a process that is simply waiting on the cpu - visible through commands such as sp_who (process will be listed as 'runnable' vs. 'running' or 'sleeping'). A good indicator for OLTP.
- *I/O load* – outstanding asynchronous I/Os. If the I/O load is fairly heavy, adding additional I/O intensive tasks such as DSS applications may be undesirable.

- *Engine deficit* – the difference in the number of online engines among instances in the cluster. Engine deficit is measurable only when instances in the cluster have unequal numbers of engines. In this scenario, engine deficit adds a metric for maximum relative capacity to the load score.
- *User metric* – a customer-supplied metric specific to the user's environment. This metric is optional.

The metric that is the most confusing is '*engine deficit*'. Since the metrics being captured are a unitless number, consider the difference in capacity between a 20 engine host and a 2 engine host when considering metrics such as the run-queue length and I/O load metrics. Assuming identical hardware otherwise, the 20 engine host could accommodate 10x the number of concurrent I/O's and concurrent processes as it has 10 times the number of engines and CPU cores available to process the pending requests. For example, if both had a metric value of 20 for the run-queue length, for the 20 engine host, the value would be almost irrelevant is it would only indicate 2 processes per engine waiting while for the 2 engine host, it translates to roughly 10 processes in queue for CPU access per engine - a fairly substantial backlog.

Initially, the workload manager collects raw metrics for each of the above. To determine whether workload distribution is required, it then normalizes the metrics according to each logical cluster's '*Load Profile*'. A load profile is simply a weighting for each of the above metrics using a 100% weighting scheme. For example, the following screen snapshot shows the weighting for the system supplied load profile 'sybase_profile_oltp':

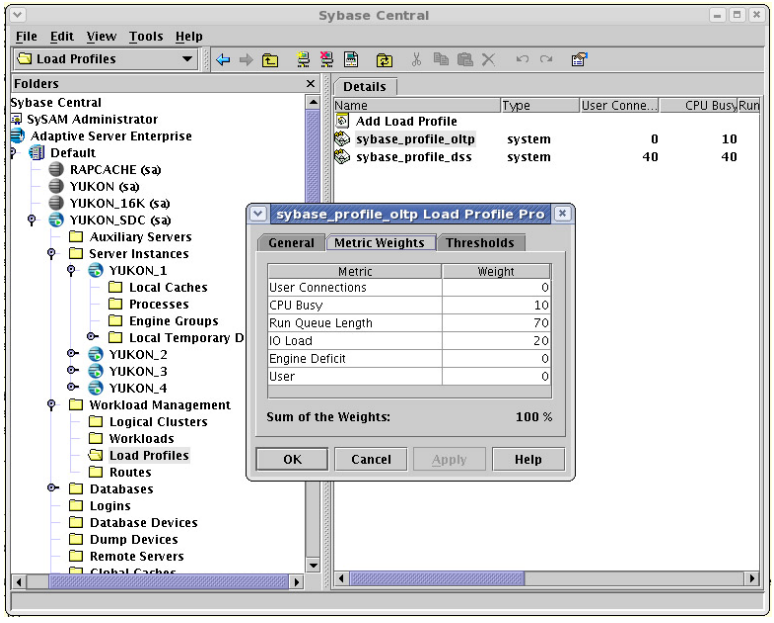


Figure 33 - Workload Metric Weighting for sybase_profile_oltp Load Profile

ASE/CE supplies two load profiles - one for OLTP and one for DSS. You can add your own load profile with different weightings as desired. As mentioned earlier, each logical cluster specifies its own load profile. The workload manager normalizes the values and then compares the workload for each of the logical clusters.

The workload manager determines if load balance is necessary by checking the '*Load Threshold*'. The load threshold is comprised of three factors:

- *Minimum Load Score* - The minimum load score necessary before any workload balancing occurs. This prevents unnecessary connection migrations between two lightly loaded instances. This minimum score is also referred to as the 'hysteresis' value.
- *Login Redirection Threshold* - The percentage difference between the instance load scores at which point login redirection for new connections starts to happen.
- *Dynamic Connection Migration Threshold* - The percentage difference between the instance load scores at which point dynamic load distribution via connection migration for existing processes starts to happen.

However, a it is important to realize that the workload manager primarily functions at the instance level and not the logical cluster level. In fact, the workload manager reports the workload at the workload profile level – which may impact multiple logical clusters. While this works fairly well in application consolidation environments, there is no direct way in ASE/CE 15.5 to provide preferences that workload distribution will occur for particular logical clusters ahead of others. The only means to control this is through using different load profile weightings - which are cluster wide and therefore hard to tailor so that workloads 'prefer' specific instances but can be distributed to other instances in a preference order fashion.

Recommendations

The following recommendations are suggested in configuring ASE/CE logical clusters and workload manager with respect to applications. Remember – an application is not a software component. For example, SAP's Business Objects is not an application – it is the software component/process in which an application executes. Similarly with WebLogic or an application server – they are simply the containers for application logic. However, this means that it may not be possible to establish the logical cluster routing by application name as few software products allow the end-user to specify the application during configuration – and fewer developers set the application name when configuring the connection pool, etc. The easiest workaround is to use the use the logical cluster as the application and ensure fairly distinctive logical cluster names with a one-to-one relationship with key application modules.

OLTP Write Intensive Applications

The primary goal of logical cluster and workload management implementations for OLTP write intensive applications is to minimize the amount of cluster overhead. Consider the following advice:

- Each logical cluster should only have a single online instance at a time.
- If the peak processing requirements exceed the capacity of a single instance, consider using application partitioning and using multiple logical clusters.

OLTP Reporting Applications

The primary goal of logical cluster and workload management implementations for OLTP reporting applications is to balance the workload while not impacting OLTP write applications.

It is important to realize, however, that there is a difference between reporting applications that can report on older data in the lifecycle vs. current data. For example, a report that does a price lookup could be considered to have minimal impact on OLTP write activities as it is pretty much unrelated. Another reporting application may report store sales for last week – which again is by nature deconflicted from current sales activity. However, a business activity monitoring application that attempts to display Key Performance Indicators (KPI) statistics in real-time will be constantly contending with the OLTP write application. Considering that categorization, consider the following advice:

- Different reporting applications should use different logical clusters
- Logical clusters for reporting applications can span more than one online instance
- Reporting applications that are unrelated to OLTP write activity or are deconflicted should run on a separate instance from OLTP write activity.
- Reporting applications that contend with write activity should either be assigned to the same logical cluster as the OLTP write application or have a separate logical cluster with the same resource assignments.

Application Partitioning: OLTP/Read-Only

Having discussed the technical challenges for cluster scalability, it makes sense to look at the different techniques for application partitioning. The first application partitioning technique is to separate reporting users from OLTP applications by moving read-only requirements to the other nodes. In a typical mixed workload application, there are any number of reports that are executed from different applications. In other cases common to customer self-service applications, a large number of web based client connections are active even though performing read-only queries that the pure CPU contention detracts from OLTP performance. In many cases, this reporting or read-only workload can be offloaded to the other nodes of the cluster.

The application partitioning for OLTP/Read-Only is the easiest to visualize:

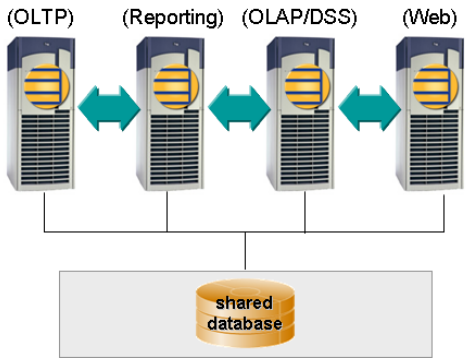


Figure 34 - Application Partitioning: OLTP vs. Read-Only

The main benefits that partitioning read-only applications to the other nodes are:

Reduced I/O contention for OLTP application. Reporting users and web users can lead to considerable amount of physical I/O requirements as they frequently are accessing uncached data pages - either due to the volume of data in the

range of the query (for reports) or the data currency as web users may look back at their transaction history.

Reduced CPU contention for OLTP application. This is especially true for reports as the typical aggregations, sorting and sheer volume of data contribute to increased CPU utilization - especially if parallel query is enabled. While web user queries are typically fairly quick, the large volume of connections and spikes of activity often lead to lengthened service request times for the OLTP application.

Improved Memory and Network Responsiveness. A side effect of the I/O contention reduction is that cache efficiency for the OLTP applications improves since current data is no longer flushed by reporting queries. This allows either systems with less memory or a closer approach to an in-memory system for OLTP query processing. In addition to the memory, network responsiveness is improved as the internal ASE network handlers and OS TCP stack is no longer deluged with large result sets common to reporting applications or the sheer volume of web user activity.

Ability to Implement Parallel Query Techniques. Parallel queries are notorious for resource consumption - both from a CPU as well as an I/O standpoint as the optimizer begins considering parallel scans as a faster alternative for index accessing.

Other benefits can be a factor such as the impact of additional security constraints on reporting/web users, the ability to run intraday reports, etc. One reason why this is often the first approach to application scalability with clustering is that it is one of the easiest to implement as the reporting applications are often separate (i.e. due to using 3rd party software such as Business Objects, Crystal Reports, Cognos, etc.) from the frontline OLTP applications. Additionally, the implementation doesn't require significant application changes the way modularizing an application or provisioning can require.

The key to scaling with OLTP vs. Read-Only reports is as follows:

- Reports that have high lock contention or frequently access the current data within minutes of it being modified should run on the same instance as the OLTP applications
- Non-critical user access (e.g. Web access by customers) should be isolated on separate instances from time-critical reporting applications. This is true even if web access allows writing (e.g. account updates, online bill pay, etc.) if the web access support is not the primary source of transactions.
- Because of the larger memory in today's systems, reporting applications should use a workload profile that considers the run-queue length the most (60%) and CPU utilization second (30%) with I/O load a distant third (10%).

The last may seem a bit weird given that reporting applications typically involve more physical IO. However, with larger memory, more reports are able to run mainly in memory with fewer physical reads per query. On the other hand, by running more in-memory, as noted before, this

will increase CPU utilization. So by focusing on outstanding IO's, you could over saturate a single instance based on CPU workload alone. The run-queue aspect would be an attempt to level the problem of a single query driving CPU and also is a way to take into consideration the number of "active" connections vs. just the flat number of "user connections".

Application Partitioning: Consolidation

Earlier, we discussed application consolidation as one of the examples for ASE/CE logical clusters. An interesting perspective is that it can provide a means of scalability in a non-traditional sense by focusing on the latency factors that often drive system response times. Consider the following generic systems infrastructure architecture common to many financial trading institutions:

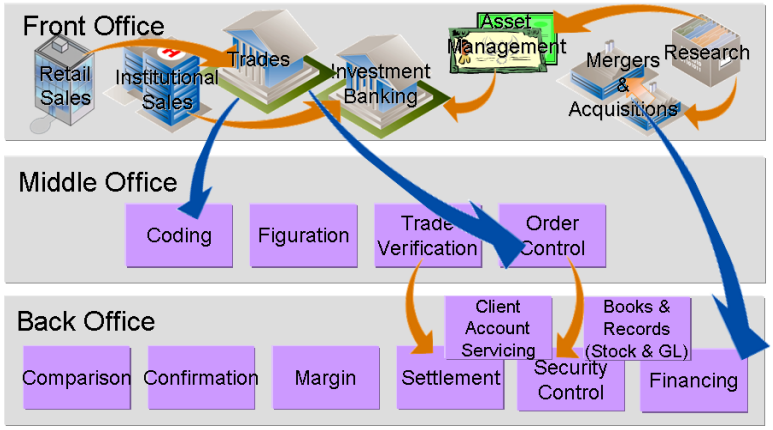


Figure 35 - Generic FSI Systems Infrastructure

For performance reasons, many of the different systems within an FSI infrastructure are highly specialized for specific business tasks - such as trade verification. However, this increases the volume of data movement between the systems these specialized systems depend on getting data feeds from other systems that precede it in the business cycle as well as need to provide data feeds to those that follow it. These data feeds can be real-time, near-real time or batch oriented depending on the timeliness of the data for the current process. Often these feeds are implemented using EAI (message buses), database replication or ETL processes.

The problem is that the latency inherent in the data movement often times drives the speed that the business can react, limits the trade volume, increases availability requirements, etc. For example, a risk system with a latency of 30 minutes based on feeds from trading systems would be limited to 30 minute old position information - which may cause a trading firm to start exiting a market later than they should have. Similarly, if it takes 2 hours to move trade verification information to the settlement system after the market closes, given the timeline to achieve trade settlement in order to avoid rejecting a trade or manual settlement, this 2 hour limits the trade volume due to the processing window that remains. In the final example, because the latency forces market close batch processes to start later than desired (i.e. the 2 hour delay for settlement), there may not be time left for any maintenance activities by the DBA's.

The reason this is a partitioning scheme is that without ASE Cluster Edition, the only alternative is to procure an extremely large SMP host that can support all the combined requirements for

the consolidated applications. From that perspective, application partitioning via consolidation allows horizontal scaling by utilizing smaller SMP systems while providing the same benefits as the larger SMP system. Application partitioning via consolidation improves the overall system performance by eliminating or reducing significantly the need for the data movements. Consider the following deployment based on the above FSI infrastructure.

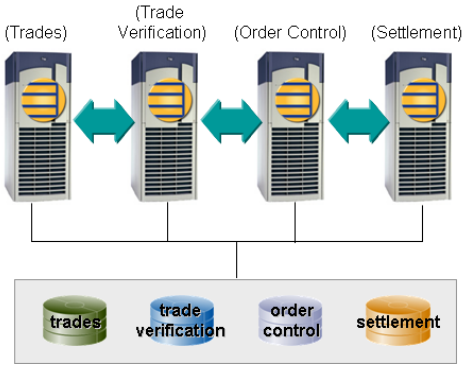


Figure 36 - Example Application Consolidation for FSI Infrastructures

The chief scalability benefit from this type of deployment has already been stated - reduction of latency by eliminating data movement between the systems. Otherwise, the other benefits are more in line with providing higher availability for the individual applications. Note that there may be some degradation of normal processing - all the cluster overhead from a code length standpoint as well as lock management may have a slight impact on normal processing. The key is whether the immediate access to upstream systems data more than offsets this degradation. In this respect, the cluster scalability takes on more a 'throughput' characteristic vs. a more traditional 'response time' characteristic as overall response times may increase but overall throughput increases as well.

The key to scaling with application consolidation is as follows:

- Each application should minimally have at least one logical cluster for OLTP activities. That logical cluster should adhere to the one active base instance guideline to prevent application contention.
- Each node of the cluster has the additional memory and CPU necessary for the cluster overhead.
- For peak processing requirements (i.e. post trade settlement), the applications that might be able to leverage more than instance should consider an additional form of application partitioning (modularization or provisioning) in addition.

If there are fewer nodes than applications, prior to consolidation, monitor the CPU utilization with monEngine to determine peak periods. Additionally, monitor active object usage via monOpenObjectActivity to get a sense of which tables are the most active. Since the applications share a lot of the same data, you then can determine better which applications/logical clusters should share the same resources. Consider the following guidelines for least impact on a per instance basis:

- Applications that modify the same data within minutes of each other should be on the same instance. Applications that modify the same data

along the lifecycle 10's of minutes to hours later can use different instances.

- Applications that share the same data and same data lifecycle currency but different peak periods.
- Applications that share the same data and same data lifecycle currency with overlapping peak periods.
- Applications that do not share the same data but have different peak periods. While this may result in cache re-warming during the beginning of each period, it should stabilize within the processing timeframe.
- Applications that do not share the same data, have overlapping peak periods, but the data cache is sufficient to support both.

Application Partitioning: Modularization

Most production schemas consist of anywhere from a few hundred to several thousand tables. Regardless of size, most schemas can be arranged around groups of tables that are often used for different transactions - often based on the business function. The application is often the same - whether a single large VisualBasic™ or PowerBuilder™ distribution or whether multiple applets, different classes of users use different "screens" than their counterparts. This is especially true from an OLTP perspective. DSS users may cross boundaries considerably, but mostly in a read-only fashion. The result is that even if only a single high volume OLTP application/database exists, it can scale horizontally by partitioning the application along the lines of application modules that are related to the user's job function.

For example, consider the very simplistic "pubs2" database shipped with ASE. From a logical perspective of any retail organization, the pubs2 database schema contains at least three different perspectives or sub-models.

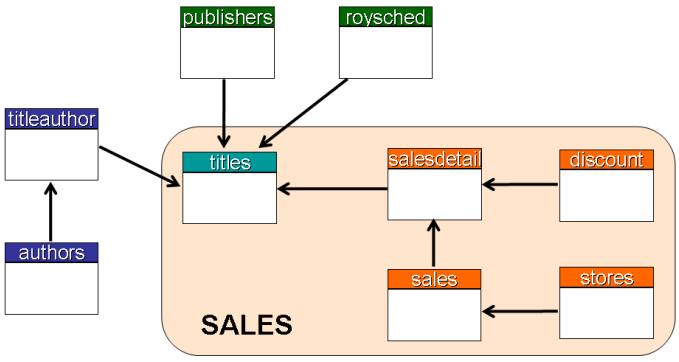


Figure 37 - Sales Perspective/Sub-Model of the Pubs2 Database

Note that in this sub-model, most of the OLTP activity is in the sales & salesdetail table - although some lookup information from the others may be needed - for example a publisher address. The 'titles' table currently is impacted as well as each insert into the salesdetail table results in an update to the total_sales column via the trigger.

While the above sub-model could be from the perspective of the sales personnel, another different perspective for the purchasing or inventory management would be:

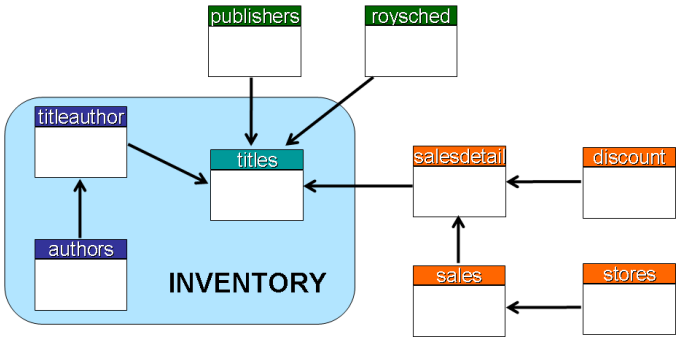


Figure 38 - Inventory (or Warehouse/Supply) Perspective Sub-Model of Pubs2 Database

In this sub-model, the OLTP activity would primarily be on inserting new titles as they are received from publishers or updating the stock-on-hand (not in the current schema).

The next sub-model could either be used by external partners (such as publishers or agents for the authors) or by the finance department to determine which books are selling in which regions, by which authors, etc.

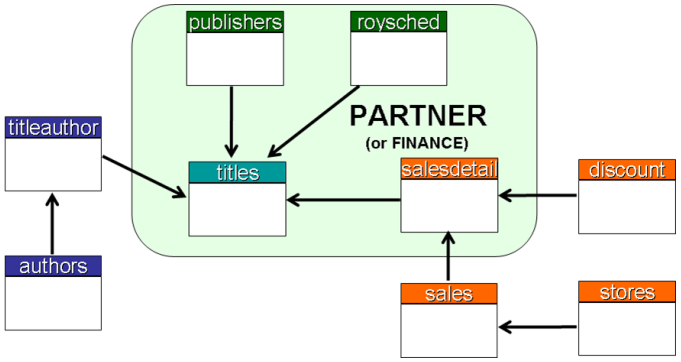


Figure 39 - Partner or Finance Perspective Sub-Model of Pubs2 Database

Note that multiple different types of users could use the same perspectives as well as a single user class may need access to multiple perspectives. For example, an inventory user may need to know that a particular store has a backorder of a title so that when it comes in, the book is shipped to that store vs. being added to the general inventory. In most cases, each user will be performing the vast majority of their DML and queries in one of the sub-models - especially considering which module of the application is in use.

Partitioning this application usually involves splitting larger application into the functional parts or collecting smaller applications and applets along functional lines such that the pieces of the application that impact the different sub-models can be distributed among the instances effectively. Note that the application will likely need to be split into different pieces and renamed or else the routing rules to direct connections to the appropriate node providing the service may not work. Consider the typical deployment of:

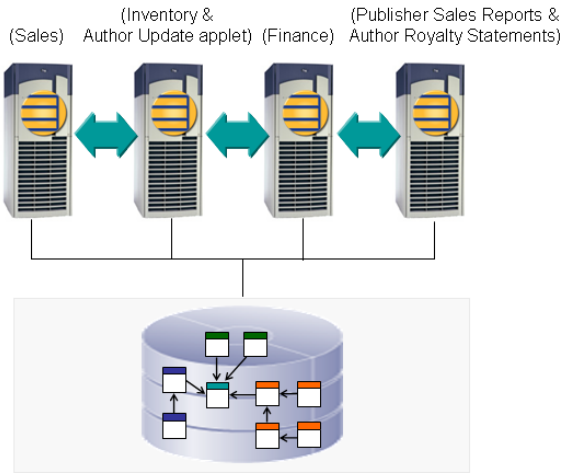


Figure 40 - Application Partitioning By Modularization

In the above illustration, authors might be using two different web applets - one to update their biography, picture, etc - the other generate royalty reports for income projections, etc. The other modular parts of the application discussed above such as Sales, Inventory and Finance/partner (Publisher Sales Reports above) are distributed according to functionality. Since the Author Update Applet affects the same sub-model as Inventory, the applet is co-located with the Inventory applet.

Obviously, the primary goal of this partitioning scheme is to improve performance and scalability for a *single* application. The question that often is raised is "What about 'titles'?" The 'titles' table will be experiencing DML activity from both the Inventory as well as the Sales application modules. The answer is dependent upon the application processing environment.

The first option is to leave the schema and application logic intact. If the inventory updates occur based on shipments received, these are likely batch updates that have singleton quantity adjustments (received 200 copies in a single statement) consequently the cache synchronization may be limited to a narrow window which may be tolerable from a performance and overhead perspective.

A second option is dependent upon how many columns of a table are modified - it may make sense to vertically partition the table if a number of columns are large enough to warrant a separate table *and* if doing so separates the DML activity.

The third option is realizing that the update to the titles.total_sales column really is out of scope for the sales portion of the transaction and could be made asynchronous. This could be implemented either by using:

- messaging between the nodes based on periodically polling new sales (vs. having each sale post a message)
- a polling process execute a replicated function via database replication to update the sales
- a periodic function use the CIS RPC mechanism to invoke a database RPC in the logical cluster where the data partition is in effect.

Once the application is divided or the applets consolidated along the sub-models, the scalability considerations are identical to those for application consolidation (above).

Application Partitioning: Provisioning

Application partitioning via provisioning is the classic horizontal scale-out of a single application that many customers find an appealing concept primarily based around the notion of replacing larger SMP hosts with cheaper, smaller machines and building as needed vs. pre-purchasing the scalability. As pointed out previously, a reasonably implied consideration is that by using commodity hardware, it can be easily and cheaply replaced. Consequently, rather than scaling out as the application grows, the capacity can be increased by replacing the older nodes with newer, more scalable nodes due to increases in CPU cores and hardware threading. While this does not address all the needs of applications needing a scale-out solution, it does provide a governing effect on the degree of scaling out necessary to achieve the desired performance requirements.

For applications that still need horizontal scaling out, the application partitioning scheme that works best is based on provisioning. The distinction is that in provisioning, the initial application workload division is pre-determined by the IT staff. For example, consider the typical partitioning schemes based on geography/site, alphabetic/numeric range, high volume customers, or other means of determining the division. Within which ever scheme, the IT staff will likely need to predetermine the number of nodes necessary and the initial provisioning of the partitioning. For example, for geography, it may be determined for maintenance or other aspects, that minimally four nodes are necessary - one each for APO, EMEA, LAO and NAO. For an alphabetic/numeric range, it might be determined that while an equal number of letters of the alphabet could be used, given the cardinality of names such as Smith or population in CA, that some nodes may have more 'letters' assigned while others have fewer due to the load. Consider the following provisioning examples:

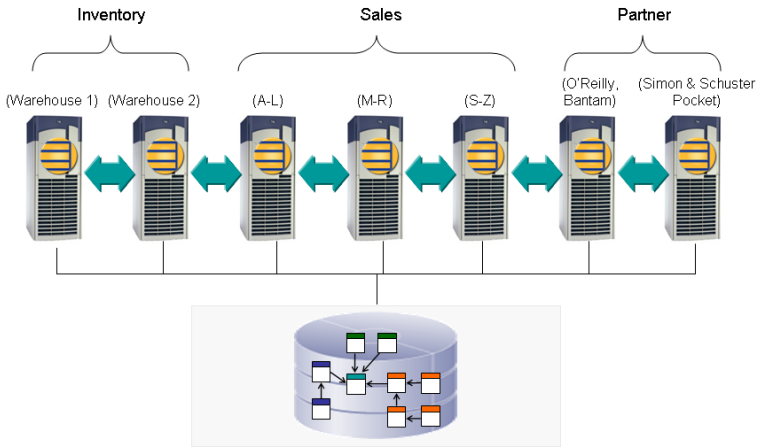


Figure 41 - Application Partitioning Using Provisioning

The two left-most nodes are provisioned by location, the middle three by alphabetic range (note the imbalance of the division based on expected cardinality in the M's, S's and T's), and the final provisioning is list based on splitting high volume publishers. Conspicuously absent is a non-provisioned partitioning scheme such as:

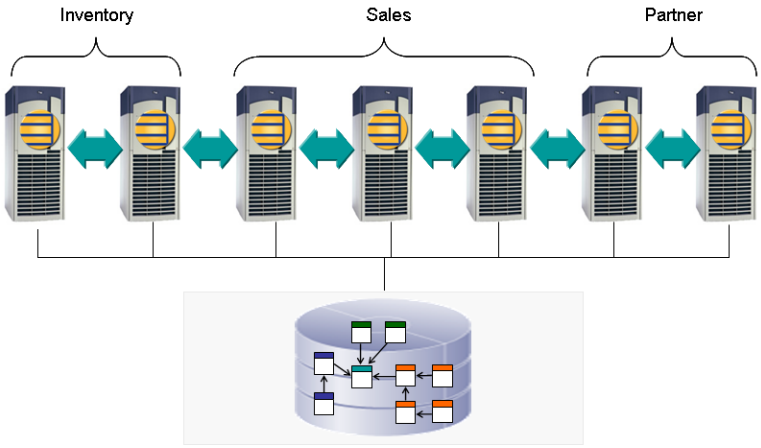


Figure 42 - Badly Partitioned Application Due to Lack of Provisioning

The reason this is badly partitioned is consider the application connection rule possibilities for any of the logical clusters (Inventory, Sales, Partner):

- Connections can be round-robin distributed
- Connections can be distributed based on a modular or hashing function on login name or similar attribute
- Connections can be distributed purely on workload

The net effect, unfortunately, is that such indiscriminant workload balancing is much more likely to result in higher cache transfers and cluster overhead. The reason is that it is unlikely that the data partitioning scheme will map evenly to the user connections (except possibly in the case of the modular or hashing if the transaction tables contain the login id or whichever connection attribute is used for hashing). A similar problem would arise if the provisioning scheme is based on the middle-tier infrastructure. For example, consider an implementation with five application servers and five nodes in the cluster. One thought might be to simply point each application server at its own node to balance the load as follows:

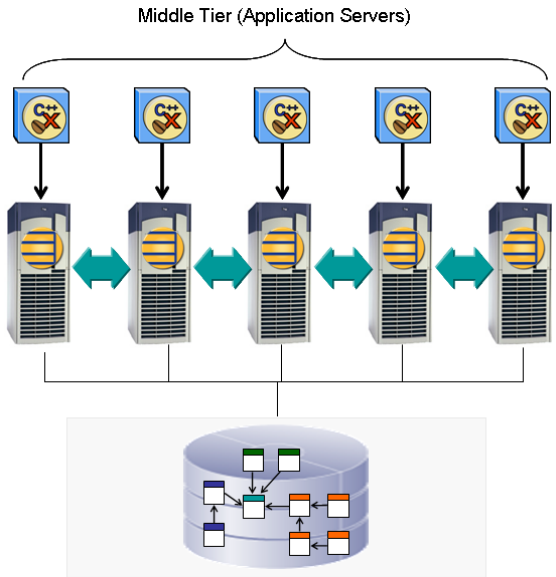


Figure 43 - Attempt to Provision Using Middle Tier

Unless the middle tier is already provisioned by location, range or list and that the data partitioning scheme matches, it would be similar to a round-robin connection mechanism in which the cache transfers would be higher than desirable. The key partitioning driver is to be able to scale out an extremely high volume application with a dominant single transaction without concern for limitations of a single platform.

The reason why partitioning needs to be based on provisioning is primary based on aligning the connections that are performing data modifications with the location of the data. This and other requirements for scalability include:

- Each application that needs to scale across multiple instances should have at least one logical cluster per instance that scalability is expected to cover during peak processing.
- The application middle-tier needs to determine how to manage the provisioning and specify the corresponding logical cluster
- The data should be physically partitioned according to the same provisioning scheme as used for the logical cluster arrangement.
- The application should maximize transaction grouping, micro-batching in addition to ensuring proper transaction scoping to prevent log contention on the single database log from becoming a scalability inhibitor.

Scalability Beyond Shared Disks

To achieve any real OLTP scalability out of a pure SDC implementation, you need to partition the application to reduce the internal contention and cross-communication between the nodes. Beyond application partitioning, the technologies that show the most promise for cluster scalability are:

Hybrid SDC/MPP IO – Hybrid SDC and MPP systems attempt to reduce some of the internal overhead within OLTP cluster scalability by assigning specific nodes to specific subsets of data. This significantly reduces the cache coherency overhead for OLTP. For OLAP and DSS systems, MPP parallelization at the disk layer attempts to provide a divide and conquer approach to the age-old problem of the heavy IO requirements of complex queries.

Data Grid Clusters – A Data Grid cluster from a DBMS parlance is more closely aligned with the virtual clusters described at the beginning of this paper in combination with the logical cluster and workload management functionality of SDC implementations.

Hybrid SDC/MPP IO

The best example of the former is the Oracle™ Database Machine with the Exadata Servers™ running underneath an Oracle RAC™ implementation. At a high level, the Exadata Servers are responsible for all physical IO requests from Oracle RAC instances. Each Exadata Server is responsible for managing a physical subset of the data in the system which requires data provisioning (or physical partitioning) of the data. Effectively, this implements MPP IO

underneath an SDC database implementation. Because MPP IO highly relies on the speed of direct attached storage (DAS) to overcome the bottleneck centralized shared storage due to low single digit GB/s HBA storage adapters, MPP IO often implements some form of data duplication that mirrors copies of the data between physical storage nodes. While Exadata cells implement data mirroring, this technique is not new and has enjoyed a rebirth in some MPP IO based DSS systems such as Vertica™ and similar implementations that use multiple copies of the data in a strategy called "K Safety" to reduce IO fault related failures as well as a technique to enable shared common data elements between IO nodes. As separate nodes IO focused nodes, Exadata cells provide query speed improvements by:

- Providing a second tier of data cache as they can cache pages requested from previous queries external to the impacts of cache management
- Reduce the amount of IO processing the DBMS nodes need to perform as search predicates can be passed through to the storage nodes to pre-filter the data pages – only returning those blocks that contain rows that match the criteria.
- Reduce the amount of CPU processing required for encrypted data by decrypting data in the storage cell.
- Reduce join processing on the DBMS by providing initial join filtering
- Cache common data aggregates (such as min/max)

One key aspect to understanding the Oracle Database Machine is that it is heavily predicated on several key facts:

- DBMS storage will still be disk centric for the future
- Large, heavily centralized infrastructure based on massive data storage is key.
- Massive hardware focus on IO to relieve key DBMS bottlenecks will be the key to scalability.
- MPP techniques such as distributed query processing will remain the key strategy for complex/large query performance.
- Row-wise data storage and access will remain key DBMS implementations.
- Large scan based reads will remain the predominant scalability inhibitor.

It attempts to mitigate the first by using ~5TB of SSD – enough to overwhelm most OLTP database requirements while providing super-fast/low latency response times for the large temporary spaces used for most analytical queries. The second point quickly becomes apparent when you consider that each Exadata cell is a Linux server with:

- 2 Intel XEON 6 core processors for a total of 12 cores per cell
- 4 96GB SSD Flash Cards for a total of 384GB per cell.
- Physical RAM not specified

For a full rack configuration, this means an Oracle Database Machine with 14 Exadata Servers is employing 168 cores just to process IO – and a good portion of that IO (384x14=4.6TB Oracle claims 5.3TB total) is exploiting SSD. By contrast, the two embedded RAC DBMS nodes contain

128 cores and 2 TB of memory. Note that the Exadata Server does nothing to accelerate write speeds outside of heavy utilization of SSD based storage – and that outside of the local Exadata Flash storage, most OLTP queries would not have much impact either due limited IO required. A good question is whether the 14 dual CPU Exadata cells could be more simply replaced with an additional 2 DBMS servers for a total of 256 cores of DBMS and 4TB of memory with appropriate software implemented partitioning.

Oracle is not alone in pursuing a SDC/MPP IO strategy. As mentioned earlier, Vertica™ has implemented a similar strategy based on data partitioning using local DAS and K Safety data duplication. Sybase IQ is not the same – in fact, it is the opposite MPP CPU/SDC IO.

Data Grid Clustering

While Oracle includes the term "Grid" in a their DBMS offerings, the best characterization is that they have implemented a SDC database on top of a storage grid. Perhaps their use of the term was driven by the similarities with grid computing. When grid computing is often visualized, many people assume that a computational grid is being implemented. In a computational grid, a complex problem is generally divided into hundreds to thousands of pieces that are executed in parallel by brute force and then the results merged.

There is a different form of grid computing that needs to be considered – the database grid. The database grid likely owes its heritage to the notion of data grids that were sometimes developed by large corporate using data virtualization layers to provide a federated data view of numerous data stores throughout the enterprise. Individual applications that relied on just a subset of the data directly accessed the line of business data in its native location. Applications that crossed lines of business accessed the data through the virtualization layer which implemented top level distributed query optimization and data caching strategies. While at some aspect this may sound very similar to the SDC/MPP IO implementation above, in the above scenario, all access is through the small number of database server instances. Those instances need to attempt to balance DBMS resources such as cache management as well as CPU scheduling among a variety of applications. The data grid cluster is based on the premise that

- The application workload service manager will intelligently route the application to the grid node that provides the matching data service – ideally on a query by query basis.
- Each data service will largely reside in-memory on the node.
- Scalability will be provided by either duplicating the data service to support read only scalability or by data partition provisioning across multiple nodes for write scalability.
- Data grid services will be provided by optimized data stores for DSS, OLTP and streaming data query processing. This will reduce the hardware footprint as well as reduce the number of distributed queries.

No mainstream DBMS vendors have implemented a data grid cluster yet – although at least one is under development.

Conclusion

In conclusion, there are a lot of features within ASE that will support scalability. With today's more powerful hardware, it is much more likely that there will either be multiple applications consolidated on the same DBMS host or competing requirements within the same application. We all know how that dedicating resources to one application typically results in underutilization of those resources when the application doesn't need them – and the inflexibility of the system to adapt to changing workloads. Ideally, in a perfect world, a pure demand based auto-tuning system would eliminate any need of pre-reserving resources or establishing resource limits for applications to avoid contention. However, often a balanced approach that marries demand-based scaling and configuration regulated resource allocation often is the best approach. Some of the key guiding principles of scalability, include:

- Isolate the workloads as much as possible to prevent one application from impacting others as much as possible.
- Isolate areas of memory to reduce cache thrashing, internal contention while retaining commonly used data values.
- Do not employ more than 10-12 instances of any feature (e.g. named caches) to avoid the feature from over complicating administration and maintenance.
- Monitor the system regularly to establish baselines for different peak business days as well as measuring the impact of changes.

In the future, SMP versions of ASE as well as cluster implementations of ASE promise even greater scalability but will likely also mean new and interesting tuning features. Some of those features may obsolete some of the guidance in this paper.

Appendix A – Server Restart Checklist

- Preserve old errorlog
- dbcc tune(des_bind)
- dbcc tune(hotdes,dbid)
- dbcc tune(deviochar)
- sp_logiosize on tempdb (check on this)
- alter database <RDDB> set durability=AT_SHUTDOWN
- add/remove dynamic listeners
- warm caches with lru table scan

Appendix B – sp_sysmon Spinlock Analysis Procedure

```
use sybsystemprocs
go

--Create #tempmonitors for proc creation
select * into #tempmonitors from master.dbo.sysmonitors where 1 = 2
go

--Create new sysmon_spinlocks procedure
if exists (select *
        from sysobjects
                where sysstat & 7 = 4
                        and name = 'sp_sysmon_spinlock')
begin
        drop procedure sp_sysmon_spinlock
end
go
print "Installing sp_sysmon_spinlock"
go


create procedure sp_sysmon_spinlock
         @NumElapsedMs int,        /* for "per Elapsed second" calculations */
         @NumXacts int             /* for per transactions calculations */
as begin

         /*
         ** SQL Script to Create SYSMON_SPINLOCK report.
         **
         ** This script creates a new section in sp_sysmon that provides spinmon
         ** like spinlock acitivity reporting directly in the sysmon report.
         **
         ** To use this script:
         **
         **       1) Install this stored procedure in the target server
         **       2) Copy definition for sp_sysmon_analyze from target server
         **                This can be gotten by using ddlgen to reverse the
         **                online copy or by locating the proc in the server's
         **                installmaster script.  Be careful - while the script
         **                largely has been unaltered, changes are not documented
         **                so relying on a single version for all target servers
         **                is not recommended.
         **       3) Add calls to this procedure as illustrated in the
         **                included sp_sysmon_analyze example
         **       4) Repeat steps 2 & 3 for sp_sysmon, adding allowance
         **                for a 'spinlock' report option as illustrated
         **                in the included sp_sysmon example
         **
         ** Code History:
         **
         ** David Wein/SYBASE              Initial development
         ** Jeff Tallman/SYBASE            Debugged looping, corrected queries,
         **                                          print formatting, added documentation
         */



         -- declare local variables
         declare  @tmp_int         int,                    -- temp var for integer storage

                  -- variables used for printing
                  @rptline         char(120),       -- formatted stats line for print statement
                  @psign           char(3),         -- hold a percent sign (%) for print out

                  -- variables used by cursors
                  @name            varchar(85),     -- spinlock name with spinlock instance id
                  @spins           int,             -- number of spins when waiting
                  @waits           int,             -- number of waits on the spinlock
                  @spin_waits      numeric(12,2),   -- ratio of spins per wait
                  @grabs           int,             -- number of times the spinlock was requested
                  @wait_percent    numeric(12,2)    -- ratio of waits to grabs and spins to waits

         /* --------- Setup Environment --------- */
         set nocount on                    -- disable row counts being sent to client

         select @psign = "%"               /* extra % symbol because '%' is escape char in print statement */

         print ""
         print ""
         print ""
         print "================================================================================"
```

```
        print ""


        -- Retrieve the spinlocks.  There are three spinlock counters collected by dbcc monitor:
        --
        --      spinlock_p_0     -> Spinlock "grabs" as in attempted grabs for the spinlock - includes waits
        --      spinlock_w_0     -> Spinlock waits - usually a good sign of contention
        --      spinlock_s_0     -> Spinlock spins - this is the CPU spins that drives up CPU utilization
        --                       The higher the spin count, the more CPU usage and the more serious
        --                       the performance impact of the spinlock on other processes not waiting
        --
        -- Note that there may be more than one spinlock per resource.  For example, with cache partitioning,
        -- there will be one spinlock per cache partition.  Spinlocks are assigned field id's as per their
        -- relative position for the group_name in master..sysmonitors.  As a result, a particular spinlock
        -- with a field_id of 5 will have a field_id of 5 for grabs, waits, and spins.  However, rather
        -- than producing cryptic output, we will calculate the offset for each instance so that each
        -- spinlock will begin with an instance of 0

        select P.field_name, rtrim(P.field_name) + "::"
                        -- calculate spinlock offset field_id to normalize to 0
                        +convert(char(7),P.field_id - (select min(field_id)
                                                       from #tempmonitors
                                                       where field_name = P.field_name)) as spinlock,
                P.value as grabs, W.value as waits, S.value as spins
          into #tempspins
          from #tempmonitors P, #tempmonitors W, #tempmonitors S
          where P.group_name = "spinlock_p_0"
            and W.group_name = "spinlock_w_0"
                    and S.group_name = "spinlock_s_0"
                    and P.field_name = W.field_name
                    and P.field_name = S.field_name
                    and W.field_name = S.field_name
                    and P.field_id = W.field_id
                    and P.field_id = S.field_id
                    and W.field_id =  S.field_id
                    -- make sure we only get spinlocks that have had observations
                    and (P.value+W.value+S.value)>0

        select comment as cache_name into #caches
          from master..syscurconfigs
          where config=19 and value > 0

        print " "
        print " "
        print "Kernel Spinlock Activity Report"
        print "-------------------------------"
        print " "


        print "Spinlock::Instance                            per sec    per xact       grabs       waits
spins/wait  contention"
        print "-------------------------------------     ----------  ----------  ----------  ----------  -----
-----  ----------"

        -- declare a cursor to get spinlocks that have only had significant waits (>1 per 100ms)
        declare kernel_cursor cursor for
                select top 15 spinlock, waits,
                        (case
                                when grabs<0 then 2147483647
                                else grabs
                                end) as grabs,
                        (case
                                when spins<0 then 2147483647
                                else spins
                                end) as spins,
                        convert(numeric(12,2),(case
                                when waits=0 then 0.0
                                when spins<0 then 2147483647/(waits*1.0)
                                else spins/(waits*1.0)
                                end)) as spin_waits,
                        convert(numeric(12,2),(case
                                when grabs=0 then 0.0
                                when grabs<0 then (100.0*waits)/2147483647
                                else (100.0*waits)/grabs
                                end)) as wait_percent
                  from #tempspins
                  where spinlock like 'Kernel->%'
-- uncomment below to filter results
--                  and ((grabs>0) and ((100.0*waits)/grabs>0))
                  order by 6 desc, 5 desc, 4 desc, 3 desc


        open kernel_cursor

        fetch kernel_cursor into @name, @waits, @grabs, @spins, @spin_waits, @wait_percent
```

```
        if (@@sqlstatus = 2)
        begin
                print "There was no significant Kernel spinlock contention in this sample interval."
        end
        else
        begin
                while (@@sqlstatus = 0)
                begin
                        select @rptline = "  " + @name + space(40 - char_length(@name)) +
                                                str(@waits / (@NumElapsedMs / 1000.0),10,1) +
                                                space(2) + str(@waits / convert(real, @NumXacts),10,1) +
                                                space(2) + str(@grabs,10,0) +
                                                space(2) + str(@waits,10,0) +
                                                space(2) + str(@spin_waits,10,1) +
                                                space(2) + str(@wait_percent,8,1) + @psign
                        print '%1!', @rptline
                        fetch kernel_cursor into @name, @waits, @grabs, @spins, @spin_waits, @wait_percent
                end
        end

        close kernel_cursor
        deallocate cursor kernel_cursor




        print " "
        print " "
        print " "
        print " "
        print "Resource Spinlock Activity Report"
        print "---------------------------------"
        print " "


        print "Spinlock::Instance                          per sec    per xact       grabs       waits
spins/wait  contention"
        print "-------------------------------------     ----------  ----------  ----------  ----------  -----
-----  ----------"

        -- declare a cursor to get spinlocks that have only had significant waits (>1 per 100ms)
        declare resource_cursor cursor for
                select top 15 spinlock, waits,
                        (case
                                when grabs<0 then 2147483647
                                else grabs
                                end) as grabs,
                        (case
                                when spins<0 then 2147483647
                                else spins
                                end) as spins,
                        convert(numeric(12,2),(case
                                when waits=0 then 0.0
                                when spins<0 then 2147483647/(waits*1.0)
                                else spins/(waits*1.0)
                                end)) as spin_waits,
                        convert(numeric(12,2),(case
                                when grabs=0 then 0.0
                                when grabs<0 then (100.0*waits)/2147483647
                                else (100.0*waits)/grabs
                                end)) as wait_percent
                  from #tempspins
                  where spinlock like 'Resource->%'
-- uncomment below to filter results
--                  and ((grabs>0) and ((100.0*waits)/grabs>0))
                  order by 6 desc, 5 desc, 4 desc, 3 desc


        open resource_cursor

        fetch resource_cursor into @name, @waits, @grabs, @spins, @spin_waits, @wait_percent

        if (@@sqlstatus = 2)
        begin
                print "There was no significant Resource spinlock contention in this sample interval."
        end
        else
        begin
                while (@@sqlstatus = 0)
                begin
                        select @rptline = "  " + @name + space(40 - char_length(@name)) +
                                                str(@waits / (@NumElapsedMs / 1000.0),10,1) +
```

```
                                                space(2) + str(@waits / convert(real, @NumXacts),10,1) +
                                                space(2) + str(@grabs,10,0) +
                                                space(2) + str(@waits,10,0) +
                                                space(2) + str(@spin_waits,10,1) +
                                                space(2) + str(@wait_percent,8,1) + @psign
                        print '%1!', @rptline
                        fetch resource_cursor into @name, @waits, @grabs, @spins, @spin_waits, @wait_percent
                end
        end

        close resource_cursor
        deallocate cursor resource_cursor




        print " "
        print " "
        print " "
        print " "
        print "Dbtable Spinlock Activity Report"
        print "--------------------------------"
        print " "


        print "Spinlock::Instance                            per sec    per xact       grabs       waits
spins/wait  contention"
        print "-----------------------------------     ----------  ----------  ----------  ----------  -----
-----  ----------"

        -- declare a cursor to get spinlocks that have only had significant waits (>1 per 100ms)
        declare dbtable_cursor cursor for
                select top 15 spinlock, waits,
                        (case
                                when grabs<0 then 2147483647
                                else grabs
                                end) as grabs,
                        (case
                                when spins<0 then 2147483647
                                else spins
                                end) as spins,
                        convert(numeric(12,2),(case
                                when waits=0 then 0.0
                                when spins<0 then 2147483647/(waits*1.0)
                                else spins/(waits*1.0)
                                end)) as spin_waits,
                        convert(numeric(12,2),(case
                                when grabs=0 then 0.0
                                when grabs<0 then (100.0*waits)/2147483647
                                else (100.0*waits)/grabs
                                end)) as wait_percent
                  from #tempspins
                  where spinlock like 'Dbtable->%' or spinlock like 'Dbt->%' or spinlock like 'Dbtable.%'
-- uncomment below to filter results
--                  and ((grabs>0) and ((100.0*waits)/grabs>0))
                  order by 6 desc, 5 desc, 4 desc, 3 desc


        open dbtable_cursor

        fetch dbtable_cursor into @name, @waits, @grabs, @spins, @spin_waits, @wait_percent

        if (@@sqlstatus = 2)
        begin
                print "There was no significant Dbtable spinlock contention in this sample interval."
        end
        else
        begin
                while (@@sqlstatus = 0)
                begin
                        select @rptline = "  " + @name + space(40 - char_length(@name)) +
                                                str(@waits / (@NumElapsedMs / 1000.0),10,1) +
                                                space(2) + str(@waits / convert(real, @NumXacts),10,1) +
                                                space(2) + str(@grabs,10,0) +
                                                space(2) + str(@waits,10,0) +
                                                space(2) + str(@spin_waits,10,1) +
                                                space(2) + str(@wait_percent,8,1) + @psign
                        print '%1!', @rptline
                        fetch dbtable_cursor into @name, @waits, @grabs, @spins, @spin_waits, @wait_percent
                end
        end

        close dbtable_cursor
```

```
        deallocate cursor dbtable_cursor




        print " "
        print " "
        print " "
        print " "
        print "Data Cache Spinlock Activity Report"
        print "-----------------------------------"
        print " "


        print "Spinlock::Instance                          per sec    per xact       grabs       waits
spins/wait  contention"
        print "-----------------------------------     ----------  ----------  ----------  ----------  -----
-----  ----------"

        -- declare a cursor to get spinlocks that have only had significant waits (>1 per 100ms)
        declare cache_cursor cursor for
                select top 15 spinlock, waits,
                        (case
                                when grabs<0 then 2147483647
                                else grabs
                                end) as grabs,
                        (case
                                when spins<0 then 2147483647
                                else spins
                                end) as spins,
                        convert(numeric(12,2),(case
                                when waits=0 then 0.0
                                when spins<0 then 2147483647/(waits*1.0)
                                else spins/(waits*1.0)
                                end)) as spin_waits,
                        convert(numeric(12,2),(case
                                when grabs=0 then 0.0
                                when grabs<0 then (100.0*waits)/2147483647
                                else (100.0*waits)/grabs
                                end)) as wait_percent
                  from #tempspins
                  where field_name in (select cache_name from #caches)
-- uncomment below to filter results
--                  and ((grabs>0) and ((100.0*waits)/grabs>0))
                  order by 6 desc, 5 desc, 4 desc, 3 desc


        open cache_cursor

        fetch cache_cursor into @name, @waits, @grabs, @spins, @spin_waits, @wait_percent

        if (@@sqlstatus = 2)
        begin
                print "There was no significant data cache spinlock contention in this sample interval."
        end
        else
        begin
                while (@@sqlstatus = 0)
                begin
                        select @rptline = "  " + @name + space(40 - char_length(@name)) +
                                                str(@waits / (@NumElapsedMs / 1000.0),10,1) +
                                                space(2) + str(@waits / convert(real, @NumXacts),10,1) +
                                                space(2) + str(@grabs,10,0) +
                                                space(2) + str(@waits,10,0) +
                                                space(2) + str(@spin_waits,10,1) +
                                                space(2) + str(@wait_percent,8,1) + @psign
                        print '%1!', @rptline
                        fetch cache_cursor into @name, @waits, @grabs, @spins, @spin_waits, @wait_percent
                end
        end

        close cache_cursor
        deallocate cursor cache_cursor

        print " "
        print " "



        print " "
        print " "
        print " "
        print " "
        print "Other Spinlock Activity Report"
```

```
        print "-----------------------------------"
        print " "


        print "Spinlock::Instance                            per sec    per xact       grabs       waits
spins/wait  contention"
        print "-------------------------------------     ----------  ----------  ----------  ----------  -----
-----  ----------"

        -- declare a cursor to get spinlocks that have only had significant waits (>1 per 100ms)
        declare other_cursor cursor for
                select top 15 spinlock, waits,
                        (case
                                when grabs<0 then 2147483647
                                else grabs
                                end) as grabs,
                        (case
                                when spins<0 then 2147483647
                                else spins
                                end) as spins,
                        convert(numeric(12,2),(case
                                when waits=0 then 0.0
                                when spins<0 then 2147483647/(waits*1.0)
                                else spins/(waits*1.0)
                                end)) as spin_waits,
                        convert(numeric(12,2),(case
                                when grabs=0 then 0.0
                                when grabs<0 then (100.0*waits)/2147483647
                                else (100.0*waits)/grabs
                                end)) as wait_percent
                  from #tempspins
                  where field_name not in (select cache_name from #caches)
                    and spinlock not like 'Dbtable->%'
                    and spinlock not like 'Dbt->%'
                    and spinlock not like 'Dbtable.%'
                    and spinlock not like 'Resource->%'
                    and spinlock not like 'Kernel->%'
-- uncomment below to filter results
--                    and ((grabs>0) and ((100.0*waits)/grabs>0))
                  order by 6 desc, 5 desc, 4 desc, 3 desc


        open other_cursor

        fetch other_cursor into @name, @waits, @grabs, @spins, @spin_waits, @wait_percent

        if (@@sqlstatus = 2)
        begin
                print "There was no significant other spinlock contention in this sample interval."
        end
        else
        begin
                while (@@sqlstatus = 0)
                begin
                        select @rptline = "  " + @name + space(40 - char_length(@name)) +
                                             str(@waits / (@NumElapsedMs / 1000.0),10,1) +
                                             space(2) + str(@waits / convert(real, @NumXacts),10,1) +
                                             space(2) + str(@grabs,10,0) +
                                             space(2) + str(@waits,10,0) +
                                             space(2) + str(@spin_waits,10,1) +
                                             space(2) + str(@wait_percent,8,1) + @psign
                        print '%1!', @rptline
                        fetch other_cursor into @name, @waits, @grabs, @spins, @spin_waits, @wait_percent
                end
        end

        close other_cursor
        deallocate cursor other_cursor

        print " "
        print " "


        return 0
end
go

exec sp_procxmode 'sp_sysmon_spinlock', 'anymode'
go
grant execute on sp_sysmon_spinlock to public



-- add the following lines to sp_sysmon_analyze (copy from $SYBASE/$SYBASE_ASE/scripts/buildmaster).  Note the
-- changes are at approximately line 260.  Lines 240-260 repeated here to help locate the placement
```

```
--

/* This stored procedure produces is used to invoke all the subordinate stored
** procedures
*/
create procedure sp_sysmon_analyze
        @interval int,
        @Reco char(1),
        @section char(80),
        @applmon char(14)
as
…(~240 lines clipped)…

                        exec @status = sp_sysmon_diskio @NumEngines, @NumElapsedMs, @NumXacts, @Reco
                if @status = 1          /* fatal error - abort run */
                begin
                                /* Subordinate stored procedures print their own messages. */
                                return 1
                end /* } */

                        exec @status = sp_sysmon_netio @NumEngines, @NumElapsedMs, @NumXacts
                        if @status = 1          /* fatal error - abort run */
                        begin /* { */
                                /* Subordinate stored procedures print their own messages. */
                                return 1
                        end /* } */
                        exec @status = sp_sysmon_repagent
                        if @status = 1          /* fatal error - abort run */
                        begin /* { */
                                /* Subordinate stored procedures print their own messages. */
                                return 1
                        end /* } */
/*
** SYSMON_SPINLOCK DIFF - ADD THIS CODE TO YOUR SYSMON_ANALYZE
*/
                        exec @status = sp_sysmon_spinlock @NumElapsedMs, @NumXacts
                        if @status = 1          /* fatal error - abort run */
                        begin
                        /* Subordinate stored procedures print their own messages. */
                        return 1
                end /* } */
/* END OF SYSMON_SPINLOCK DIFF */
…
```

CONTACT INFORMATION

For Europe, Middle East,
or Africa inquiries:
+(31) 34 658 2999

For Asia-Pacific inquiries:
+852 2506 8900 (Hong Kong)

For Latin America inquiries:
+770 777 3131 (Atlanta, GA)

Sybase, an SAP Company
Worldwide Headquarters
One Sybase Drive
Dublin, CA 94568-7902 USA
Tel: +800 8 SYBASE

www.sybase.com

SYBASE®
An SAP Company