Performance and Tuning Series:
Basics

Adaptive Server® Enterprise

15.7

DOCUMENT ID: DC20020-01-1570-01

LAST REVISED: September 2011

Contents

CHAPTER 1 Introduction to the Basics

Good performance

Performance is the measure of efficiency for an application or for multiple applications running in the same environment. Performance is usually measured in **response time** and **throughput**.

Response time

Response time is the number of milliseconds, seconds, minutes, hours, or days that a single task takes to complete. You can improve response times by:

- Making queries, transactions, and batches more efficient through query tuning and indexes
- Using faster components (for example, faster client and server processors, and faster disks and storage).
- Minimizing wait times (for example, by improving network, physical, and logical lock contention)

In some cases, Adaptive Server® is automatically optimized to reduce initial response time, that is, the time it takes to return the first row to the user. This is especially useful when a user retrieves several rows with a query and then uses a front-end tool to browse them slowly.

Throughput

Throughput refers to the volume of work completed per unit of time. For example, the amount of work performed by:

- The number of a single transactions (for example, 100 transactions per second inserting trades from Wall Street).
- All transactions across the server (for example, 10,000 read transactions per second plus 1,500 write transactions per second).
- The number of reads performed (for example, the number of specific queries or reports per hour).

However, when you tune Adaptive Server for improved response times, you may decrease throughput, and vice versa. For example:

- Adding indexes improves performance for queries and updates and deletes that use those indexes to avoid more expensive scans. However, you must maintain indexes during data manipulation language (DML) operations, which can decrease performance.
- Using synchronous disk writes improves response times in a single transaction that includes a single user, but synchronous disk writes degrade multiuser throughput.

Designing for performance

Most performance gains derive from good database design, thorough query analysis, and appropriate indexing. You can realize the largest performance gains by establishing a good database design and working with the Adaptive Server query optimizer as you develop your applications.

You can also improve performance by analyzing how an application works with Adaptive Server. For example, a client may initially send rows with a size of 1KB to Adaptive Server and then wait for Adaptive Server to acknowledge receiving the rows before it sends the next row. You may find that the performance between the client and Adaptive Server improves if the client consolidates or batches the rows it sends to Adaptive Server, greatly simplifying the process and requiring less interaction between Adaptive Server and the client.

You can also use hardware and network analysis, to locate performance bottlenecks in your installation.

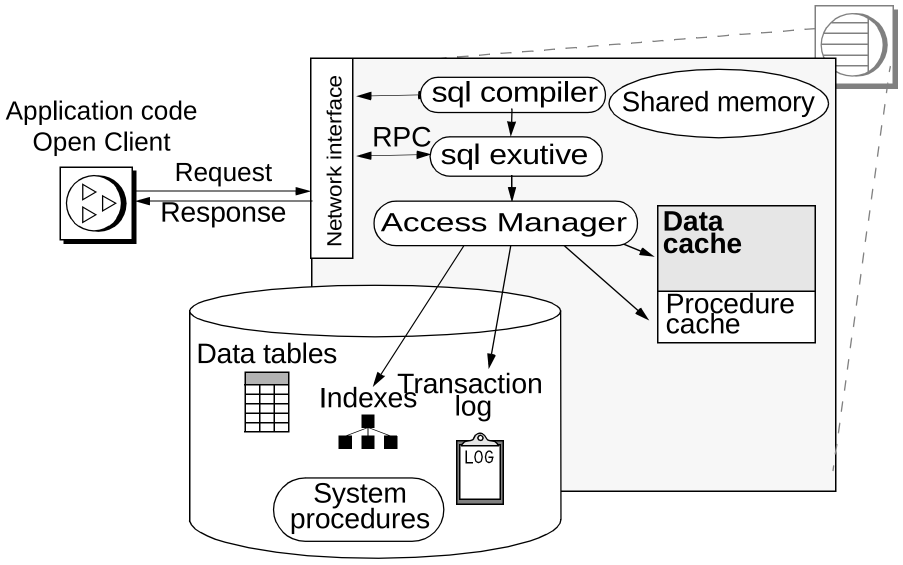
Tuning performance

Tuning improves performance and decreases contention and resource consumption. System administrators view:

- System tuning – tuning the system as a whole. See *Performance and Tuning Series: Physical Database Tuning*.
- Query tuning – making queries and transactions faster, and making the logical and physical database design more efficient. See *Performance and Tuning Series: Query Processing and Abstract Plans*.

Use this system model of Adaptive Server and its environment to identify performance problems at each layer.

Figure 1-1: Adaptive Server system model



A major part of system tuning is reducing contention for system resources. As the number of users increases, contention for resources such as data and procedure caches, spinlocks on system resources, and the CPUs increases. The probability of logical lock contention also increases.

Tuning levels

Adaptive Server and its environment and applications can be broken into components, or tuning layers, to isolate components of the system for analysis. In many cases, you must tune two or more layers so that they work optimally together.

In some cases, removing a resource bottleneck at one layer reveals another problem area. More optimistically, resolving one problem sometimes alleviates other problems. For example, if physical I/O rates are high for queries, and you add more memory to speed response time and increase your cache hit ratio, you may ease problems with disk contention.

Application layer

Most performance gains come from query tuning, based on good database design. At the application layer, these issues are relevant:

- Decision-support systems (DSS) and online transaction processing (OLTP) require different performance strategies.
- Transaction design can reduce performance, since long-running transactions hold locks and reduce access to data.
- Relational integrity requires joins for data modification.
- Indexing to support selects increases the time required to modify data.
- Auditing for security purposes can limit performance.

Options to address these issues include:

- Remote or replicated processing to move decision support off the OLTP machine
- Stored procedures that reduce compilation time and network usage
- The minimum locking level that meets application needs

Database layer

Applications share resources at the database layer, including disks, the transaction log, and data cache.

One database may have 2^{31} (2,147,483,648) logical pages. These logical pages are divided among devices, up to the limit available on each device. Therefore, the maximum possible size of a database depends on the number and size of available devices.

"Overhead" is space reserved to the server, and is not available for any user database. The overhead is calculated by summing the:

- Size of the master database, plus
- The size of the model database, plus
- The size of tempdb, plus
- (For Adaptive Server version 12.0 and later) the size of sybsystemdb, plus
- 8KB for the server's configuration area.

At the database layer, issues that affect overhead include:

- Developing a backup and recovery scheme
- Distributing data across devices
- Running auditing
- Efficiently scheduling maintenance activities that can slow performance and lock users out of tables

Address these issues by:

- Automating log dumps with transaction log thresholds to avoid space problems
- Monitoring space with thresholds in data segments
- Adding partitions to speed loading of data and query execution
- Distributing objects across devices to avoid disk contention or to take advantage of I/O parallelism
- Caching for high availability of critical tables and indexes

Adaptive Server layer

At the server layer, there are many shared resources, including the data and procedure caches, thread pools, locks, and CPUs.

Issues at the Adaptive Server layer include:

- The application types to be supported: OLTP, DSS, or a mix.
- The number of users to be supported —as the number of users increases, contention for resources can shift.
- Number of threads in the thread pool.
- Network loads.

- Replication Server® or other distributed processing can be an issue when the number of users and transaction rate reach high levels.

Address these issues by:

- Tuning memory (the most critical configuration parameter) and other parameters
- Deciding some processing can take place at the client side
- Configuring cache sizes and I/O sizes
- Reorganizing the thread pools
- Adding multiple CPUs
- Scheduling batch jobs and reporting for off-hours
- Reconfiguring certain parameters for shifting workload patterns
- Determining whether DSS and move to another Adaptive Server

Devices layer

The devices layer relates to the disk and controllers that store your data. Adaptive Server can manage a virtually unlimited number of devices.

Issues at the devices layer include:

- The distribution of system databases, user databases, and database logs across devices
- Whether partitions for parallel query performance or high insert performance on heap tables are necessary

Address these issues by:

- Using more medium-sized devices and controllers; doing so may provide better I/O throughput than a few large devices
- Distributing databases, tables, and indexes to create even I/O load across devices

- Using segments and partitions for I/O performance on large tables used in parallel queries

---

Note Adaptive Server devices are mapped to operating system files or raw partitions, and depend on the performance of the underlying physical devices and disks. Contention can occur at the operating system level: controllers can be saturated and the disks organized under storage area network logical unit numbers (SAN LUNs) can be over-worked. When analyzing performance, balance the physical load to properly distribute load over operating system devices, controller, storage entities, and logical unit numbers.

Network layer

The network layer relates to the network or networks that connect users to Adaptive Server.

Virtually all Adaptive Server users access their data via a network.

Issues at this layer include:

- The amount of network traffic
- Network bottlenecks
- Network speed

Address these issues by:

- Configuring packet sizes to match application needs
- Configuring subnets
- Isolating heavy network uses
- Moving to a higher-capacity network
- Designing applications to limit the amount of network traffic required

Hardware layer

The hardware layer concerns the CPUs and memory available.

Issues at the hardware layer include:

- CPU throughput
- Disk access: controllers as well as disks

- Disk backup
- Memory usage
- Virtual machine configuration: resource allotment and allocation

Address these issues by:

- Adding CPUs to match workload
- Configuring the housekeeper tasks to improve CPU utilization
- Following multiprocessor application design guidelines to reduce contention
- Configuring multiple data caches

Operating system layer

Ideally, Adaptive Server is the only major application on a machine, and must share CPU, memory, and other resources only with the operating system and other Sybase® software such as Backup Server™.

Issues the operating system layer include:

- The file systems available to Adaptive Server
- Memory management – accurately estimating operating system overhead and other program memory use
- CPU availability and allocation to Adaptive Server

Address these issues by:

- Considering the network interface
- Choosing between files and raw partitions
- Increasing the memory size
- Moving client operations and batch processing to other machines
- Using multiple CPUs for Adaptive Server

Identifying system limits

The physical limits of the CPU, disk subsystems, and networks impose performance limits. Latency or bandwidth limitations are imposed by device driver, controllers, switches, and so on. You can overcome some of these limits by adding memory, using faster disk drives, switching to higher bandwidth networks, and adding CPUs.

Threads, thread pools, engines and CPUs

In process mode, Adaptive Server typically consumes one CPU per configured engine. In threaded mode, Adaptive Server typically consumes one CPU per configured engine thread, plus additional CPU for nonengine threads, such as I/O handling threads in syb_system_pool.

However, the definition of a CPU is ambiguous in modern systems. What the operating system reports as a CPU may be a core of a multicore processor or a subcore thread, where each core supports multiple threads (often called hyperthreading, SMT, or CMT). For example, a system that has two 4-core processors with two threads per core, reports that it has 16 CPUs. This does not mean that all 16 CPUs are available for Adaptive Server.

Sybase recommends that you determine how much actual CPU is available to Adaptive Server, and how much CPU power each engine and each nonengine thread requires. A good estimate is to allow one CPU for the operating system. In threaded mode, also allow one CPU for I/O threads.

Based on this recommendation, configure no more than 15 engines on a 16-CPU system in process mode, and no more than 14 engines in threaded mode on the same system (in threaded mode, each engine can do more useful work than in process mode).

When configuring the CPU, consider that:

- Servers with high I/O loads may require that you reserve more CPU for I/O threads.
- Not all CPUs are equal,and you may not be able to reserve all subcore threads. For example, you may need to treat an 8-core, 16-thread system as if it has only 12 CPUs.
- You may be required to reserve CPU for other applications on the host.

Sybase recommends that you use sp_sysmon to validate your configuration. If you see a high degree of nonvoluntary context switching, or an engine tick utilization that is higher than the OS thread utilization, you may have over-configured Adaptive Server relative to the underlying CPU, which can lead to a significant loss of throughput.

Varying logical page sizes

The dataserver binary builds the master device (located in *$SYBASE/ASE-15_0/bin*). The dataserver command allows you to create master devices and databases with logical pages of size 2KB, 4KB, 8KB, or 16KB. Larger logical pages can provide benefits for some applications:

- You can create longer columns and rows than with smaller page sizes, allowing wider tables.
- Depending on the nature of your application, you may improve performance, since Adaptive Server can access more data each time it reads a page. For example, reading a single 16K page brings 8 times the amount of data into cache as reading as a 2K page; reading an 8K page brings in twice as much data as a 4K page, and so on.

  However, when you use larger pages, queries that access only one row in a table (called point queries) use rows that occupy more memory. For example, each row you save to memory in a server configured for 2k logical pages uses 2k, but if the server is configured for 8k logical pages, each row you save to memory uses 8k.

  Analyze individual cases to verify whether using logical pages larger than 2KB is beneficial.

Number of columns and column size

The maximum number of columns you can create in a table is:

- Fixed-length columns in allpages-locked and data-only-locked tables – 1024.
- Variable-length columns in an allpages-locked table – 254.
- Variable-length columns in a data-only-locked table – 1024.

The maximum size of a column depends on:

- Whether the table includes variable- or fixed-length columns.
- The logical page size of the database. For example, in a database with 2K logical pages, the maximum size of a column in an allpages-locked table can be as large as a single row, about 1962 bytes, less the row format overheads. Similarly, for a 4K page, the maximum size of a column in an allpages-locked table can be as large as 4010 bytes, less the row format overheads.

Maximum length of expressions, variables, and stored procedure arguments

The maximum size for expressions, variables, and arguments passed to stored procedures is 16384 (16K) bytes, for any page size for character or binary data. You can insert variables and literals up to this maximum size into text columns without using the writetext command.

Number of logins

Table 1-1 lists the limits for the number of logins, users, and groups for Adaptive Server.

Table 1-1: Limits for number of logins, users, and groups

Item	Version 15.0 limit
Number of logins per server (SUID)	2147516416
Number of users per database	2146585223
Number of groups per database	1032193

Performance implications for limits

The limits set for Adaptive server mean that the server may have to handle large volumes of data for a single query, DML operation, or command. For example, if you use a data-only-locked table with a char(2000) column, Adaptive Server must allocate memory to perform column copying while scanning the table. Increased memory requests during the life of a query or command mean a potential reduction in throughput.

Size of kernel resource memory

The kernel resource memory is a cache. Adaptive Server reserves the kernel resource memory as a pool from which all thread pools receive their memory. The maximum size you can allocate to the kernel resource memory is 2147483647 2K logical pages.

Analyzing performance

When you have performance problems, determine the sources of the problems and your goals in resolving them.

To analyze performance problems:

1. Collect performance data to get baseline measurements. For example, you might use one or more of the following tools:
   - Internally developed benchmark tests or industry-standard third-party tests.
   - sp_sysmon, a system procedure that monitors Adaptive Server performance and provides statistical output describing the behavior of your Adaptive Server system.

     See *Performance and Tuning Series: Monitoring Adaptive Server with sp_sysmon*.
   - Monitoring tables, which describe resource utilization and contention from a server-wide to a user- or object-level perspective.
   - Any other appropriate tools.
2. Analyze the data to understand the system and any performance problems. Create and answer a list of questions to analyze your Adaptive Server environment. The list might include questions such as:
   - What are the symptoms of the problem?
   - What components of the system model affect the problem?
   - Does the problem affect all users or only users of certain applications?
   - Is the problem intermittent or constant?
3. Define system requirements and performance goals:
   - How often is this query executed?

- What response time is required?

4. Define the Adaptive Server environment – know the configuration and limitations at all layers.
5. Analyze application design – examine tables, indexes, and transactions.
6. Formulate a hypothesis about possible causes of the performance problem and possible solutions, based on performance data.
7. Test the hypothesis by implementing the solutions from the last step:
   - Adjust configuration parameters.
   - Redesign tables.
   - Add or redistribute memory resources.
8. Use the same tests used to collect baseline data in step 1 to determine the effects of tuning. Performance tuning is usually a repetitive process.

   If the actions taken based on step 7 do not meet the performance requirements and goals set in step 3, or if adjustments made in one area cause new performance problems, repeat this analysis starting with step 2. You may need to reevaluate system requirements and performance goals.
9. If testing shows that your hypothesis is correct, implement the solution in your development environment.

www.sybase.com includes whitepapers that discuss additional ways to analyze performance.

Normal forms

Normalization is an integral part of the relational database design process and can be employed to reorganize a database to minimize and avoid inconsistency and redundancy.

The different normal forms organizes administrator information so that it promotes efficient maintenance, storage, and data modification. Normalization simplifies query and update management, including the security and integrity of the database. However, normalization usually creates a larger number of tables which may, in turn, increase the size of the database.

Database designers and administrators must decide on the various techniques best suited their environment.

Locking

Adaptive Server locks the tables, data pages, or data rows currently used by active transactions to protect them. Locking is necessary in a multiuser environment, since several users may be working with the same data at the same time.

Locking affects performance when one process holds locks that prevent another process from accessing data. The process that is blocked by the lock sleeps until the lock is released. This is called lock contention.

A more serious locking impact on performance arises from deadlocks. A deadlock occurs when two user processes each have a lock on a separate page or table and each wants to acquire a lock on the same page or table held by the other process. The transaction with the least accumulated CPU time is killed and all of its work is rolled back.

Understanding the types of locks in Adaptive Server can help you reduce lock contention and avoid or minimize deadlocks.

The *Performance and Tuning Series: Locking and Concurrency Control* discusses the performance implications of locking.

Special considerations

When you create a database in Adaptive Server, you can assign its storage to one or more data storage devices (see Chapter 7, “Initializing Database Devices in *System Administration Guide: Volume 1*). Information about these devices is stored in master.dbo.sysdevices. Declare which device to use for the database, and how much of each device this database uses. A database can occupy all available space on the device, or other databases can share space on the device, or any combination of the two. Segments (logical groupings of storage within a database) allow you to keep some data logically or physically separate from other data. For example, to aid in disaster recovery, Sybase strongly recommends that you physically separate the transaction log from other data within a database.

Logical and physical data groupings can help your database perform better. For example, you can reserve part of the database for a data set that you know will grow much larger over time by assigning this data set, and no other, a particular segment. You can also physically separate heavily used indexes from their data to help prevent disk "thrashing," which slows down read and write response times.

Note For Adaptive Server, devices provide a logical map of a database to physical storage, while segments provide a logical map of database objects to devices. To achieve your space allocation goals, it is important that you understand the interplay between these logical layers.

Each database can have up to 32 named segments. Adaptive Server creates and uses three of these segments:

- system segment– contains most system catalogs.
- default segment – used if you do not specify one when creating an object.
- logsegment – stores the transaction log.

You can store user tables in the system segment, but the logsegment is reserved entirely for the log.

Adaptive Server keeps track of the various pieces of each database in master.dbo.sysusages. Each entry in sysusages describes one fragment of a database. Fragments are a contiguous group of logical pages, all on the same device, that permit storage for the same group of segments. Fragments are also known as "disk pieces."

Because of the way Adaptive Server allocates and maintains database space, these disk fragments are even multiples of 256 logical pages, which is one **allocation unit**. When you decide how large to make a device, consider the number of allocation units that are required, since the device size should be evenly divisible by the allocation unit size (256 times the logical page size). If it is not, the space at the end of that device is wasted, since Adaptive Server cannot allocate it. For example, if your server uses a 16K page, then one allocation unit is 4MB (16K times 256). If you create a device of 103MB on that server, the last 3 MB cannot be allocated and are wasted.

Note The master device is an exception to this rule. The master device is the first device you create when you install a new server. Adaptive Server reserves 8K of space at the beginning of the master device for a configuration area that is not part of any database. Take this space into account when you create your master device. 8K is 0.0078125MB (about .008MB). You will waste the least space in your master device if, for example, you specify 200.008MB, as its size, rather than 200MB.

A database cannot be larger than 2,147,483,648 pages. The logical page size determines the number of bytes: using a 2K page, it is 4 terabytes, on a 16K page Adaptive Server, it is 32 terabytes.

You can divide the storage for your databases between devices in any way you want. The theoretical limit for the number of disk fragments per database is 8,388,688. However, the practical limit depends on the Adaptive Server memory configuration. To use a database, Adaptive Server must hold the database’s storage description in memory. This includes a description of the database’s “disk map,” which includes all the disk fragments you have assigned storage to for the database. In practice, a database’s storage complexity is limited by the amount of memory configured for Adaptive Server, and is not normally a problem.

However, databases with disk maps that contain thousands of disk fragments may pay a penalty in performance. When Adaptive Server needs to read or write a page, it converts the page’s logical page number to a location on disk by looking it up in the disk map. Although this lookup is fast, it does take time, and the amount of time gets longer as you add more disk fragments to the map.

CHAPTER 2 Networks and Performance

This chapter discusses the role that networks play in the performance of applications using Adaptive Server.

Usually, the system administrator is the first to recognize a problem on the network or in performance, including such things as

- Process response times vary significantly for no apparent reason.
- Queries that return a large number of rows take longer than expected.
- Operating system processing slows down during normal Adaptive Server processing periods.
- Adaptive Server processing slows down during certain operating system processing periods.
- A particular client process seems to slow all other processes.

Potential performance problems

Some of the underlying problems that can be caused by networks are:

- Adaptive Server uses network services inefficiently.
- The physical limits of the network have been reached.
- Processes retrieve unneeded data values, which unnecessarily increase network traffic.

- Processes open and close connections too often, increasing network load.
- Processes frequently submit the same SQL transaction, causing excessive and redundant network traffic.
- Adaptive Server does not have enough network memory.
- Adaptive Server network packet sizes are not big enough to handle the type of processing needed by certain clients.

Basic questions on network performance

When looking at network-related problems, ask yourself these questions:

- Which processes usually retrieve a large amount of data?
- Are a large number of network errors occurring?
- What is the overall performance of the network?
- What is the mix of transactions being performed using SQL and stored procedures?
- Are a large number of processes using the two-phase commit protocol?
- Are replication services being performed on the network?
- How much of the network is being used by the operating system?

Techniques summary

Once you have gathered the data, you can take advantage of several techniques that should improve network performance, including:

- Using small packets for most database activity
- Using larger packet sizes for tasks that perform large data transfers
- Using stored procedures to reduce overall traffic
- Filtering data to avoid large transfers
- Isolating heavy network users from ordinary users
- Using client control mechanisms for special cases

Use sp_sysmon while making network configuration changes to observe the effects on performance. See *Performance and Tuning Series: Monitoring Adaptive Server with sp_sysmon.*

Engine and thread affinity

When configured for threaded mode, Adaptive Server tasks have a limited affinity to a specific engine.

Network listeners

A network listener is a system task that listens on a given network port for incoming client connections, and creates one database management system task for each client connection. Adaptive Server creates one listener task for each network port on which Adaptive Server listens for incoming client connection requests. Initially, these ports consist of the master entries in the interfaces file.

The initial number of network listener tasks is equal to the number of master entries in the interfaces file. The maximum number of network listeners (including those created at start-up) is 32. For example, if there are two master entries in the interfaces file under the server name at startup, you can create 30 more listener tasks.

Each additional listener task that you create consumes resources equal to a user connection. So, after creating a network listener, Adaptive Server can accept one less user connection. The number of user connections configuration parameter includes both the number of network listeners and the number of additional listener ports.

The number of listener ports is determined at startup by the number of master entries in the interfaces file.

For more information about the interfaces file, see Chapter 1, “Overview of System Administration,” in the *System Administration Guide: Volume 1*.

Network listeners in process mode

In process mode, each Adaptive Server engine is a separate process; therefore, network listeners run slightly differently than they do in threaded mode, when Adaptive Server is a single process.

In process mode:

- Adaptive Server uses one listener task per port. Each listener task functions as multiple logical listeners by switching from engine to engine, attempting to balance the load. For example, a 64-engine Adaptive Server with two master ports has two listener tasks, but these two listener tasks act as 128 logical listener tasks, so the server has two physical and 128 logical listeners. Starting a listener on engine 3 does not result in Adaptive Server spawning a new listener task unless the port does not already have a listener
- A listener task accepts connections on engines on which it is enabled. So a single listener task corresponds to many logical listeners.
- Stopping a listener on a specific engine terminates the logical listener for this engine since the listener task no longer switches to that engine. Adaptive Server terminates the listener task in case this was the last engine on which it was allowed to operate.

How Adaptive Server uses the network

All client/server communication occurs over a network, by way of packets. Packets contain a header and routing information, as well as the data they carry.

Clients initiate a connection to the server. The connection sends client requests and server responses. Applications can have as many connections open concurrently as they need to perform the required task.

The protocol used between the client and server is known as the Tabular Data Stream™ (TDS), which forms the basis of communication for many Sybase products.

Configuring the I/O controller

Adaptive Server includes I/O controllers and an I/O controller manager.

The I/O controller issues, tracks, polls, and completes I/Os. Each Adaptive Server I/O type—disk, network, Client-Library, and, for the Cluster Edition, CIPC—has its own I/O controller.

Adaptive Server can include multiple instances of disk or network controllers (multiple CIPC or Client-Library controllers are not allowed). Each task represents one controller. For example, configuring three network tasks means network I/O uses three controllers.

Each controller task is allocated a dedicated operating system thread. Additional tasks mean more CPU resources are dedicated to polling and completing I/O.

A single I/O controller per system is usually sufficient. However, you may need additional controllers on systems with very high I/O rates or low single-thread performance. In this situation, engines may become starved for I/O, and throughput decreases.

Use the sp_sysmon "Kernel Utilization" section to determine if additional I/O tasks are necessary.

Consider additional I/O tasks if:

- The "Thread Utilization (OS %)" for the I/O tasks exceeds the "Engine Busy Utilization".
- The "Thread Utilization (OS %)" for the I/O tasks exceeds 50%.
- The polls returning "max events" in the Controller Activity section is greater than zero.
- The "average events per poll" in the Controller Activity section is greater than three.

The mode for which you configure Adaptive Server determines how it handles I/O. In threaded mode—the default—Adaptive Server uses threaded polling for I/O; in process mode, Adaptive Server uses a polling scheduler for I/O.

In process mode, Adaptive Server assigns each engine its own network, disk, and Open Client controller. When the scheduler polls for I/O, it searches only the engine's local controllers (except for CIPC, for which all engines share a single controller).

One benefit to process mode polling is that, when you scale the number of engines, you scale the amount of CPU available to manage I/Os (that is, more engines = more CPU). However, you can configure too much CPU to manage the I/Os, devoting more time to a higher number of engines than is necessary to perform the tasks. Another performance implication is that the engine on which the I/O starts must finish the I/O (that is, if a task running on engine 2 issues a disk I/O, that I/O must be completed by engine 2, even if other engines are idle). This means that engines may remain idle even while there are tasks to perform, and I/Os may incur additional latency if the responsible engine is running a CPU-bound task.

When configured for threaded polling, the controller manager assigns each controller a task, and this task is placed into syb_system_pool. Because syb_system_pool is a dedicated pool, it creates a thread to service each task. This thread runs the polling and completion routine exclusively for the I/O controller. Because this thread is dedicated to performing this task, the task can block waiting for I/O completions, reducing the number of system calls and empty polls.

You can create multiple threads to service each I/O controller, allowing you to avoid single-thread saturation problems during which a single thread cannot keep up with a high rate of I/Os.

Process-mode polling introduces I/O latency when the I/O completes at the operating system level. However, the engine does not detect I/O latency because the engine is running another task. Threaded-mode polling eliminates this latency because the I/O thread task processes the completion immediately, and any I/O latency is a function of the device, and is not affected by the CPU load the query thread execution places on the system.

In threaded mode, the query processor and user tasks need not context switch for I/O polling when they go through the scheduler. Threaded polling reduces the amount of time spent polling as a percentage of total CPU time for all threads, making Adaptive Server more efficient in CPU consumption.

Use sp_configure with number of disk tasks and number of network taks to determine the number of tasks dedicated to handling I/O and the thread polling method the tasks use.

See Chapter 5, “Setting Configuration Parameters,” in *System Administration Guide, Volume 1*.

By default, each I/O task uses a thread from syb_system_pool, allowing the task to block during the I/O polling, reducing overhead from busy polling. During periods of low I/O load, these threads consume little physical CPU time. The CPU time for the I/O thread increases as the I/O load increases, but the amount of load increase depends on the processor performance and the I/O implementation.

Dynamically reconfiguring I/O tasks

When you increase the number of I/O tasks, there may be a slight lag before Adaptive Server balances the load across tasks. When you increase the number of disk I/Os, Adaptive Server quickly balances the distribution across the controllers. However, network tasks have an affinity between the connection and the task, so when you increase the number of network tasks, they are not rebalanced across the new higher number of tasks. Instead, Adaptive Server rebalances the load as existing connections disconnect and new connections are made.

You must restart Adaptive Server to reduce the number of I/O tasks.

Changing network packet sizes

Typically, OLTP sends and receives large numbers of packets that contain very little data. A typical insert or update statement may be only 100 or 200 bytes. A data retrieval, even one that joins several tables, may bring back only one or two rows of data, and still not completely fill a packet. Applications using stored procedures and cursors also typically send and receive small packets.

Decision-support applications often include large query batches and return larger result sets.

In both OLTP and DSS environments, there may be special needs, such as batch data loads or text processing, that can benefit from larger packets.

For most applications, the default network packet size of 2048 works well. Change the default network packet size to 512 if the application uses only short queries and receives small result sets.

Chapter 5, “Setting Configuration Parameters,” in *System Administration Guide: Volume 1* describes how to change these configuration parameters:

- The default network packet size
- The max network packet size and additional network memory, which provide additional memory space for large packet connections

Only a system administrator can change these configuration parameters.

Large versus default packet sizes for user connections

Adaptive Server reserves enough space for all configured user connections to log in at the default packet size. Large network packets cannot use that space. Connections that use the default network packet size always have three buffers reserved for the connection.

Connections that request large packet sizes acquire the space for their network I/O buffers from the additional network memory region. If there is not enough space in this region to allocate three buffers at the large packet size, connections use the default packet size instead.

Number of packets is important

Generally, the number of packets being transferred is more important than the size of the packets. Network performance includes the time needed by the CPU and operating system to process a network packet. This per-packet overhead has the most effect on performance. Larger packets reduce the overall overhead costs and achieve higher physical throughput, provided that you have enough data to be sent.

The following big transfer sources may benefit from large packet sizes:

- Bulk copy
- readtext and writetext commands
- select statements with large result sets
- Insertions that use larger row sizes

There is always a point at which increasing the packet size stops improving performance, and may, in fact, decrease performance, because the packets are not always full. Although there are analytical methods for predicting that point, it is more common to vary the size experimentally and plot the results. If you conduct such experiments over a period of time and a variety of conditions, you can determine a packet size that works well for many processes. However, since the packet size can be customized for every connection, you may also want to conduct specific experiments for specific processes.

Results can vary significantly between applications. You may find that bulk copy works best at one packet size, while large image data retrievals perform better at a different packet size.

If testing shows that some applications can achieve better performance with larger packet sizes, but that most applications send and receive small packets, clients request the larger packet size.

Adaptive Server evaluation tools

The sp_monitor system procedure reports on packet activity. This report shows only the packet-related output:

```
...
packets received   packets sent   packet errors
----------------   ------------   --------------
10866(10580)       19991(19748)             0(0)
...
```

You can also use these global variables:

- *@@pack_sent* – number of packets sent by Adaptive Server.
- *@@pack_received* – number of packets received.
- *@@packet_errors* – number of errors.

These SQL statements show how you can use these counters:

```
select "before" = @@pack_sent
select * from titles
select "after" = @@pack_sent
```

Both sp_monitor and the global variables report all packet activity for all users since the last restart of Adaptive Server.

See Chapter 14 “Using Batches and Control-of-Flow Language,” in the *Transact-SQL Users Guide* for more information about sp_monitor and these global variables.

Other evaluation tools

Operating system commands also provide information about packet transfers. See your operating system documentation.

Server-based techniques for reducing network traffic

Using stored procedures, views, and triggers can reduce network traffic. These Transact-SQL tools can store large chunks of code on the server so that only short commands need to be sent across the network.

- Stored procedures – applications that send large batches of Transact-SQL commands may place less load on the network if the SQL is converted to stored procedures. Views can also help reduce the amount of network traffic.

  You may be able to reduce network overhead by turning off doneinproc packets.
- Ask for only the information you need – applications should request only the rows and columns they need, filtering as much data as possible at the server to reduce the number of packets that need to be sent. In many cases, this can also reduce the disk I/O load.
- Large transfers – simultaneously decrease overall throughput and increase the average response time. If possible, perform large transfers during off-hours. If large transfers are common, consider acquiring network hardware that is suitable for such transfers. Table 2-1 shows the characteristics of some network types.

Table 2-1: Network options

Type	Characteristics
Token ring	Token ring hardware responds better than Ethernet hardware during periods of heavy use.
Fiber optic	Fiber-optic hardware provides very high bandwidth, but is usually too expensive to use throughout an entire network.
Separate network	Use a separate network to handle traffic between the highest volume workstations and Adaptive Server.

- Network overload – network managers rarely detect problems before database users start complaining to their system administrator.

  Be prepared to provide local network managers with predicted or actual network requirements when they are considering adding resources. Also, monitor the network and try to anticipate problems that result from newly added equipment or application requirements.

Impact of other server activities

You should be aware of the impact of other server activity and maintenance on network activity, especially:

- Two-phase commit protocol
- Replication processing
- Backup processing

These activities, especially replication processing and the two-phase commit protocol, involve network communication. Systems that make extensive use of these activities may see network-related problems. Accordingly, try to perform these activities only as necessary. Try to restrict backup activity to times when other network activity is low.

Single user versus multiple users

You must take the presence of other users into consideration before trying to solve a database problem, especially if those users are using the same network.

Since most networks can transfer only one packet at a time, many users may be delayed while a large transfer is in progress. Such a delay may cause locks to be held longer, which causes even more delays.

When response time is abnormally high, and normal tests indicate no problem, it could be due to other users on the same network. In such cases, ask the user when the process was being run, if the operating system was generally sluggish, if other users were doing large transfers, and so on.

In general, consider multiuser impacts, such as the delay caused by a long transaction, before digging more deeply into the database system to solve an abnormal response time problem.

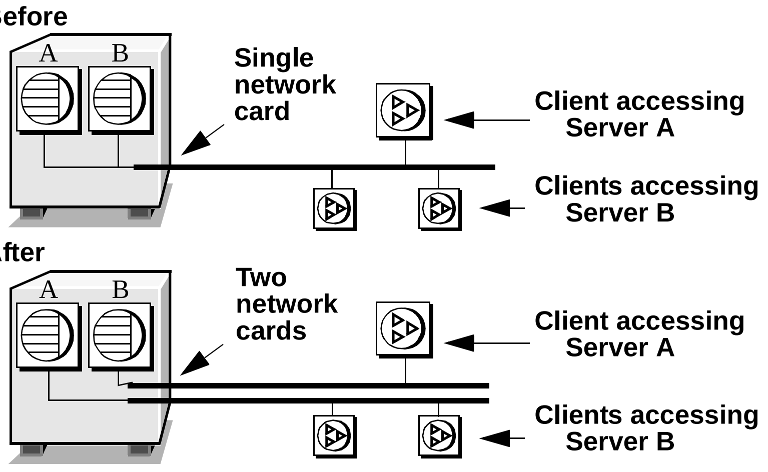
Improving network performance

There are several ways you may be able to improve network performance.

Isolate heavy network users

Isolate heavy network users from ordinary network users by placing them on a separate network, as shown in Figure 2-1.

Figure 2-1: Isolating heavy network users


Before
A
B
Single network card
Client accessing Server A
Clients accessing Server B
After
A
B
Two network cards
Client accessing Server A
Clients accessing Server B

In the "Before" diagram, clients accessing two different Adaptive Servers use one network card. Clients accessing Servers A and B must compete over the network and past the network card.

In the "After" diagram, clients accessing Server A use one network card and clients accessing Server B use another.

Set *tcp no delay* on TCP networks

By default, tcp no delay is set to on, meaning that packet batching is disabled.

When tcp no delay is set to off, the network batches packets, briefly delaying the dispatch of partial packets over the network. While this improves network performance in terminal-emulation environments, it can slow performance for Adaptive Server applications that send and receive small batches. To enable packet batching, set tcp no delay to 0, or off.

Configure multiple network listeners

Use two (or more) ports listening for a single Adaptive Server. Direct front-end software to any configured network port by setting the DSQUERY environment variable.

Using multiple network ports spreads out the network load and eliminates or reduces network bottlenecks, thus increasing Adaptive Server throughput.

See the *Adaptive Server Configuration Guide* for your platform for information on configuring multiple network listeners.

CHAPTER 3 Using Engines and CPUs

The Adaptive Server multithreaded architecture is designed for high performance in both uniprocessor and multiprocessor systems. This chapter describes how Adaptive Server uses engines and CPUs to fulfill client requests and manage internal operations. It introduces Adaptive Server's use of CPU resources, describes the Adaptive Server symmetric multiprocessing (SMP) model, and illustrates task scheduling with a processing scenario.

This chapter also gives guidelines for multiprocessor application design and describes how to measure and tune CPU- and engine-related features.

Background concepts

A relational database management system (RDBMS) must be able to respond to the requests of many concurrent users. An RDBMS must also maintain its transaction state while ensuring all transactional properties. Adaptive Server is based on a multithreaded, single-process architecture that manages thousands of client connections and multiple concurrent client requests without overburdening the operating system.

In a system with multiple CPUs, enhance performance by configuring Adaptive Server to use multiple Adaptive Server engines. In threaded kernel mode (the default), each engine is an operating system thread. In process mode, each engine is a separate operating system process.

All engines are peers that communicate through shared memory as they act upon common user databases and internal structures such as data caches and lock chains. Adaptive Server engines service client requests. They perform all database functions, including searching data caches, issuing disk I/O read and write requests, requesting and releasing locks, updating, and logging.

Adaptive Server manages the way in which CPU resources are shared between the engines that process client requests. It also manages system services (such as database locking, disk I/O, and network I/O) that impact processing resources.

How Adaptive Server processes client requests

Adaptive Server creates a new client task for every new connection. This is how it fulfills a client request:

1. The client program establishes a network socket connection to Adaptive Server.
2. Adaptive Server assigns a task from the pool of tasks, which are allocated at start-up time. The task is identified by the Adaptive Server process identifier, or spid, which is tracked in the sysprocesses system table.
3. Adaptive Server transfers the context of the client request, including information such as permissions and the current database, to the task.
4. Adaptive Server parses, optimizes, and compiles the request.
5. If parallel query execution is enabled, Adaptive Server allocates subtasks to help perform the parallel query execution. The subtasks are called worker processes, which are discussed in the *Performance & Tuning Series: Query Processing and Abstract Plans*.
6. Adaptive Server executes the task. If the query was executed in parallel, the task merges the results of the subtasks.
7. The task returns the results to the client, using TDS packets.

For each new user connection, Adaptive Server allocates a private data storage area, a dedicated stack, and other internal data structures.

Adaptive Server uses the stack to keep track of each client task's state during processing, and uses synchronization mechanisms such as queueing, locking, semaphores, and spinlocks to ensure that only one task at a time has access to any common, modifiable data structures. These mechanisms are necessary because Adaptive Server processes multiple queries concurrently. Without these mechanisms, if two or more queries were to access the same data, data integrity would be compromised.

The data structures require minimal memory resources and minimal system resources for context-switching overhead. Some of these data structures are connection-oriented and contain static information about the client.

Other data structures are command-oriented. For example, when a client sends a command to Adaptive Server, the executable query plan is stored in an internal data structure.

Client task implementation

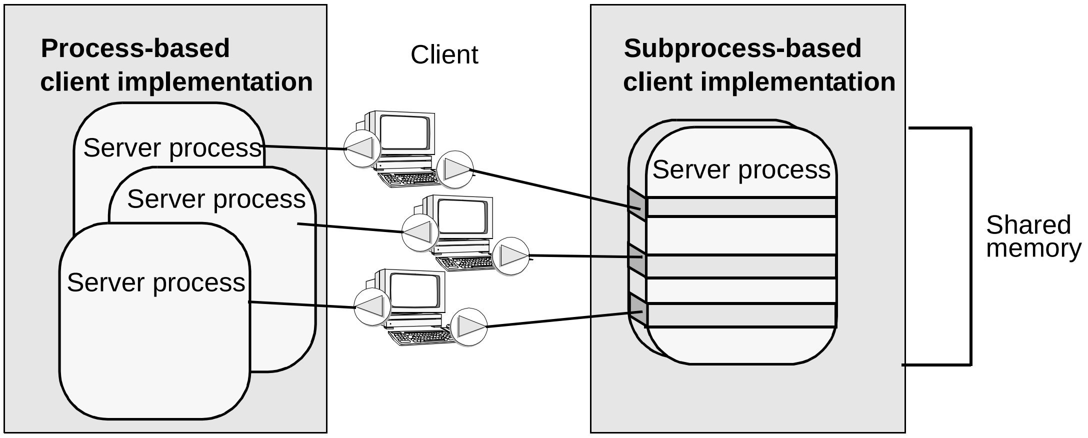
Adaptive Server client tasks are implemented as subprocesses, or "lightweight processes," instead of operating system processes. Subprocesses use only a small fraction of the resources that processes use.

Multiple processes executing concurrently require more memory and CPU time than multiple subprocesses. Processes also require operating system resources to switch context from one process to the next.

Using subprocesses eliminates most of the overhead of paging, context switching, locking, and other operating system functions associated with a one-process-per-connection architecture. Subprocesses require no operating system resources after they are launched, and they can share many system resources and structures.

Figure 3-1 illustrates the difference in system resources required by client connections implemented as processes and client connections implemented as subprocesses. Subprocesses exist and operate within a single instance of the executing program process and its address space in shared memory.

Figure 3-1: Process versus subprocess architecture



To give Adaptive Server the maximum amount of processing power, run only essential non-Adaptive Server processes on the database machine.

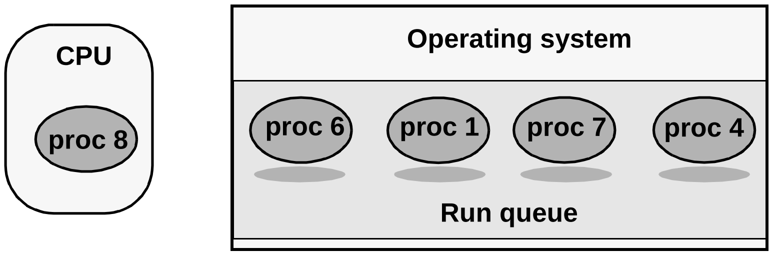
Single-CPU process model

In a single-CPU system, Adaptive Server runs as a single process, sharing CPU time with other processes, as scheduled by the operating system.

Scheduling engines to the CPU

Figure 3-2 shows a run queue for a single-CPU environment in which process 8 (proc 8) is running on the CPU and processes 6, 1, 7, and 4 are in the operating system run queue waiting for CPU time. Process 7 is an Adaptive Server process; the others can be any operating system process.

Figure 3-2: Processes queued for a single CPU

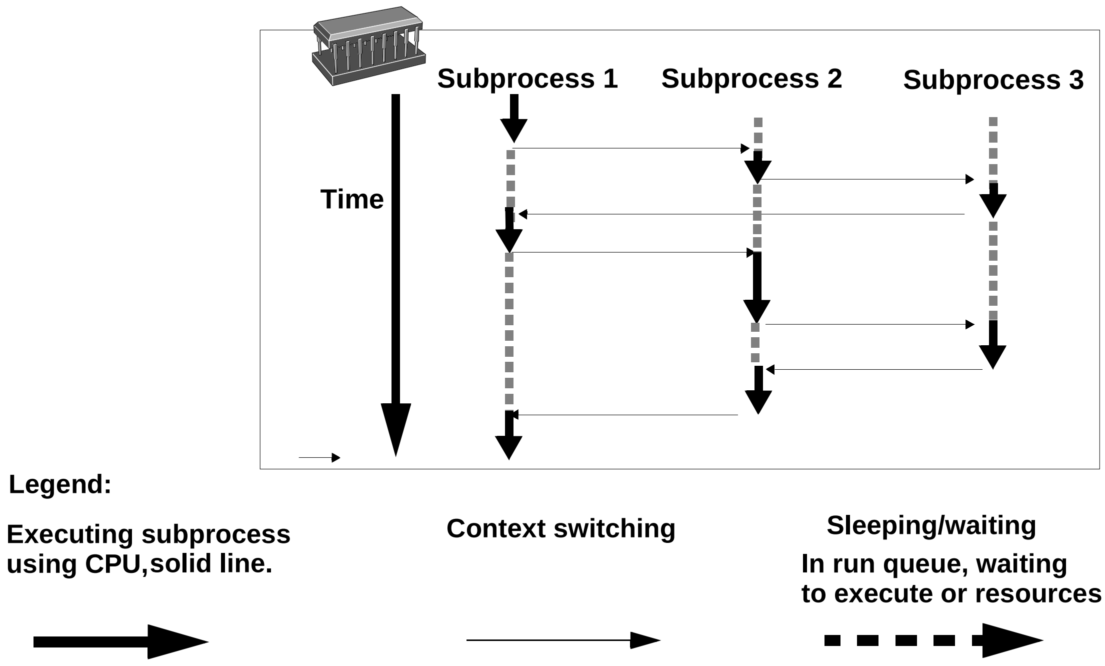


In a multitasking environment, multiple processes or subprocesses execute concurrently, alternately sharing CPU resources.

Figure 3-3 shows three subprocesses in a multitasking environment. The subprocesses share a single CPU by switching onto and off the engine over time.

At any one time, only one process is executing. The other processes sleep in various stages of progress.

Figure 3-3: Multithreaded processing


Subprocess 1
Subprocess 2
Subprocess 3
Time
Legend:
Executing subprocess using CPU,solid line.
Context switching
Sleeping/waiting
In run queue, waiting to execute or resources

Scheduling tasks to the engine

Figure 3-4 shows internally processing for tasks (or worker processes) queued for an Adaptive Server engine in a single-CPU environment. Adaptive Server, not the operating system, dynamically schedules client tasks from the run queue onto the engine. When the engine finishes processing one task, it executes the task at the beginning of the run queue.

When a task begins running, the engine continues processing it until one of the following events occurs:

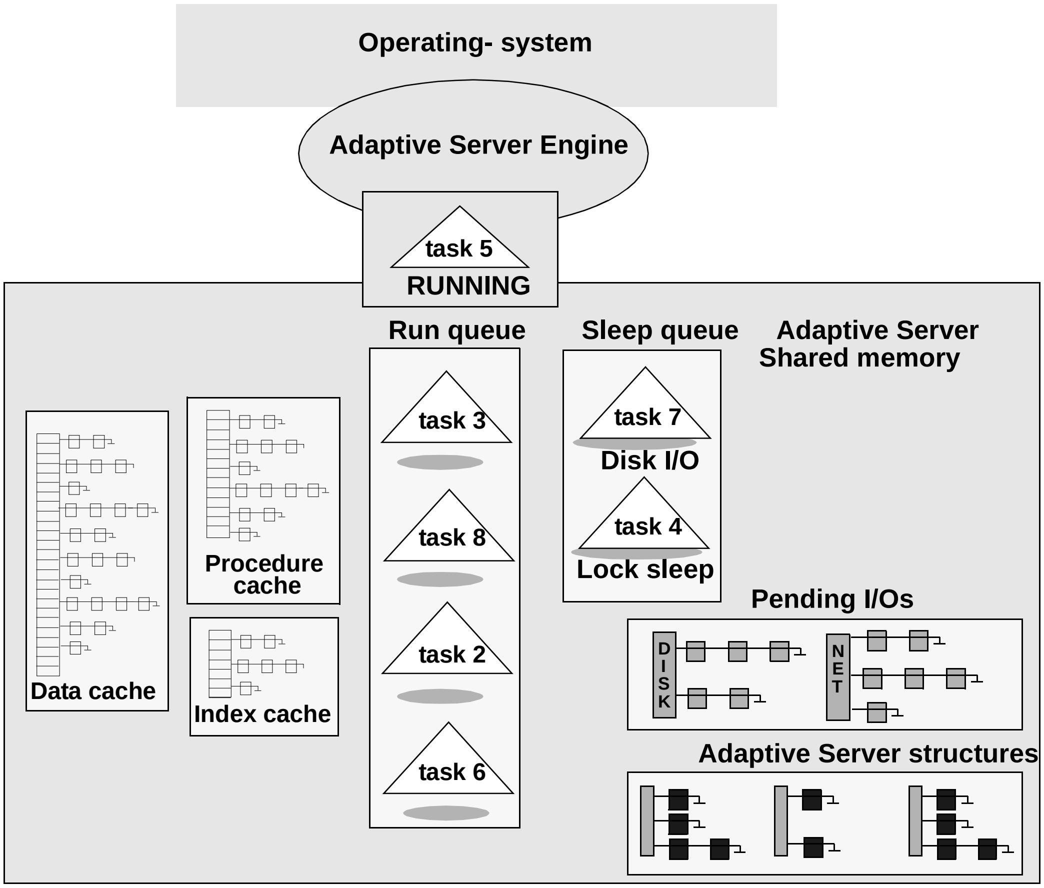
- The task completes, returns data (if any), metadata and statuses to the client. When the task completes, it appears in the sp_sysmon section `Task Context Switches Due To` as `Network Packet Received.`
- If an Adaptive Server engine does not find any runnable tasks, it can either relinquish the CPU to the operating system or continue to look for a task to run by looping for the number of times set by runnable process search count.

  Adaptive Server engines attempt to remain scheduled for the processor as long as possible. The engine runs until it is emptied by the operating system. However, if there is insufficient work available, the engine checks I/O and runnable tasks.
- The task runs for a configurable period of time and reaches a yield point `Voluntary Yields` in sp_sysmon). The task relinquishes the engine, and the next process in the queue starts to run. “Scheduling client task processing time” on page 38 discusses in more detail how this works.

When you execute sp_who on a single-CPU system with multiple active tasks, sp_who output shows only a single task as “running”—it is the sp_who task itself. All other tasks in the run queue have the status “runnable.” The sp_who output also shows the cause for any sleeping tasks.

Figure 3-4 also shows the sleep queue with two sleeping tasks, as well as other objects in shared memory. Tasks are put to sleep while they wait for resources or for the results of a disk I/O operation.

Figure 3-4: Tasks queue up for the Adaptive Server engine



Execution task scheduling

The scheduler manages processing time for client tasks and internal housekeeping.

Scheduling client task processing time

The time slice configuration parameter prevents executing tasks from monopolizing engines during execution. The scheduler allows a task to execute on an Adaptive Server engine for a maximum amount of time equal to the time slice and cpu grace time values combined, using default times for time slice (100 milliseconds, 1/10 of a second, or equivalent to one clock tick) and cpu grace time (500 clock ticks, or 50 seconds).

Adaptive Server scheduler does not force tasks off an Adaptive Server engine. Tasks voluntarily relinquish the engine at a yield point, when the task does not hold a vital resource such as a spinlock.

Each time the task comes to a yield point, it checks to see if time slice has been exceeded. If it has not, the task continues to execute. If execution time does exceed time slice, the task voluntarily relinquishes the engine. However, if the task does not yield even after exceeding time slice, Adaptive Server terminates the task after it exceeds cpu grace time. The most common cause for a task not yielding is a system call that does not return in a timely manner.

For more information about using sp_sysmon to determine how many times tasks yield voluntarily, see “Scheduling tasks to the engine” on page 36.

To increase the amount of time that CPU-intensive applications run on an engine before yielding, assign execution attributes to specific logins, applications, or stored procedures.

If the task has to relinquish the engine before fulfilling the client request, the task goes to the end of the run queue, unless there are no other tasks in the run queue. If no tasks are in the queue when an executing task reaches a yield point during grace time, Adaptive Server grants the task another processing interval.

Normally, tasks relinquish the engine at yield points prior to completion of the cpu grace time interval. It is possible for a task not to encounter a yield point and to exceed the time slice interval. If time slice is set too low, an engine may spend too much time switching between tasks, which tends to increase response time. If time slice is set too high, CPU-intensive processes may monopolize the CPU, which can increase response time for short tasks. If your applications encounter timeslice errors, adjusting the value for time slice has no affect, but adjusting the value for cpu grace time does. However, research the cause of the time slice error before adjusting the value for cpu grace time. You may need to contact Sybase Technical Support.

When the cpu grace time ends, Adaptive Server terminates the task with a time slice error. If you receive a time slice error, try increasing the time up to four times the current time for cpu grace time. If the problem persists, call Sybase Technical Support.

See Chapter 4, “Distributing Engine Resources.”

Maintaining CPU availability during idle time

The idle timout parameter for create thread pool and alter thread pool determines the amount of time, in microseconds, a thread in this pool looks for work before going to sleep. You can set idle timout for engine pools only, not for RTC pools. See “Setting Configuration Parameters” in the *System Administration Guide: Volume 1*.

The default for idle timout is 100 microseconds. However, Adaptive Server may not precisely honor the timeout period, especially at lower values (lower than 100).

Once you set the value for idle timeout, Adaptive Server registers the value in the configuration file under the Thread Pool heading:

```
[Thread Pool:new_pool]
     number of threads = 1
     idle timeout = 500
```

Setting idle timeout to -1 prevents the engines from yielding. At this setting, the engines consume 100% of the CPU.

Note The idle timeout parameter replaces the runnable process search count configuration parameter used in versions of Adaptive Server earlier than 15.7. idle timeout is available for threaded mode only. However, runnable process search count remains available for process mode.

Adaptive Server SMP process model

Adaptive Server’s symmetric multiprocessing (SMP) implementation extends the performance benefits of Adaptive Server’s multithreaded architecture to multiprocessor systems. In the SMP environment, multiple CPUs cooperate to perform work faster than a single processor can.

SMP is intended for machines with the following features:

- A symmetric multiprocessing operating system
- Shared memory over a common bus

- Two to 1024 processors (128 processors in process mode)
- Very high throughput

Scheduling engines to CPUs

The symmetric aspect of SMP is a lack of affinity between processes and CPUs—processes are not attached to a specific CPU. Without CPU affinity, the operating system schedules engines to CPUs in the same way as it schedules non-Adaptive Server processes to CPUs. Scheduling any process to a processor, including Adaptive Server engines, is done by the operating system, which can, at any time, preempt an Adaptive Server engine to run an arbitrary task. If an Adaptive Server engine does not find any runnable tasks, it can either relinquish the CPU to the operating system or continue to look for a task to run according to the amount of time specified by the idle timeout parameter.

In some situtations,you may improve performace by forcing an association between Adaptive Server threads and a specific CPU or set of CPUs. For example, grouping engines into the fewest number of physical sockets improves the hit rate on the L2 and L3 caches, improving performance.

In configurations where a single socket has sufficient parallelism for all engine and I/O threads (such as an 8-core socket running a 4-engine Adaptive Server), consider binding the Adaptive Server engine to a single socket with dbcc tune or with your operating system (generally recommended). Consult your operating system documentation for instructions on binding threads or processes to CPUs.

Scheduling Adaptive Server tasks to engines

Scheduling Adaptive Server tasks to engines in the SMP environment is similar to scheduling tasks in the single-CPU environment, as described in “Scheduling tasks to the engine” on page 36. However, in the SMP environment:

- Each engine has a run queue. Tasks have soft affinities to engines. When a task runs on an engine, it creates an affinity to the engine. If a task yields the engine and then is queued again, it tends to be queued on the same engine’s run queue.

- Any engine can process the tasks in the global run queue, unless logical process management has been used to assign the task to a particular engine or set of engines.
- When an engine looks for a task to execute, it first looks in the local and global run queues, then in run queues for other engines, where it steals a task with appropriate properties.:

Multiple network engines

In process mode, when a user logs in to Adaptive Server, the task is assigned in round-robin fashion to one of the engines is serving as the task's network engine. This engine establishes the packet size, language, character set, and other login settings. All network I/O for a task is managed by its network engine until the task logs out.

In threaded mode, any engine can issue network I/O for any task. Network polling is performed by the dedicated network tasks in the syb_system_pool.

Task priorities and run queues

Adaptive Server may increase the priority of some tasks, especially if they are holding an important resource or have had to wait for a resource. In addition, logical process management allows you to assign priorities to logins, procedures, or applications using sp_bindexeclass and related system procedures.

See Chapter 4, "Distributing Engine Resources," for more information on performance tuning and task priorities.

Each task has a priority assigned to it; the priority can change over the life of the task. When an engine looks for a task to run, it first scans its own high-priority queue and then the high-priority global run queue.

If there are no high-priority tasks, it looks for tasks at medium priority, then at low priority. If it finds no tasks to run on its own run queues or the global run queues, it can examine the run queues for another engine, and steal a task from another engine. This combination of priorities, local and global queues, and the ability to move tasks between engines when workload is uneven provides load balancing.

Tasks in the global or engine run queues are all in a runnable state. Output from sp_who lists tasks as "runnable" when the task is in any run queue.

Processing scenario

These steps describe how a task is scheduled in the SMP environment. The execution cycle for single-processor systems is very similar. A single-processor system handles task switching, putting tasks to sleep while they wait for disk or network I/O, and checking queues in the same way.

1. In process mode, when a connection logs in to Adaptive Server, it is assigned to a task that manages its network I/O.

   The task assigns the connection to an engine or engine group and establishes packet size, language, character set, and other login settings. A task sleeps while waiting for the client to send a request.

2. Checking for client requests.

   In process mode, another task checks for incoming client requests once every clock tick.

   In threaded mode, Adaptive Server wakes a dedicated thread as soon as a new request arrives.

   When this second task finds a command (or query) from the connection, it wakes up the first task and places it on the end of its run queue.

3. Fulfilling a client request.

   When a task becomes first in the queue, the query processor parses, compiles, and begins executing the steps defined in the task's query plan.

4. Performing disk I/O.

   If the task needs to access a page locked by another user, it is put to sleep until the page is available. After such a wait, the task's priority is increased, and it is placed in the global run queue so that any engine can run it.

5. Performing network I/O.

   In threaded mode, because there is no network affinity, tasks return results from any engine.

   In process mode, if a task is executing on its network engine, the results are returned. If the task is executing on an engine other than its network engine, the executing engine adds the task to the network engine's high-priority queue.

Asynchronous log service

Asynchronous log service, or ALS, enables scalability in Adaptive Server, providing higher throughput in logging subsystems for high-end symmetric multiprocessor systems.

For every database on which ALS is enabled, one engine predominantly performs log I/O, so ALS is beneficial only when the contention on the log semaphore is higher than the processing power of one engine.

You cannot use ALS if you have fewer than four engines.

Enabling ALS

Use sp_dboption to enable, disable, or configure ALS

```
sp_dboption database_name, "async log service", "true|false"
```

checkpoint (which writes all dirty pages to the database device) is automatically executed as part of sp_dboption.

This example enables ALS for mydb:

```
sp_dboption "mydb", "async log service", "true"
```

Disabling ALS

Before you disable ALS, make sure there are no active users in the database. If there are, you receive an error message.

This example disables ALS:

```
sp_dboption "mydb", "async log service", "false"
```

Displaying ALS

Use sp_helpdb to see whether ALS is enabled in a specified database:

```
sp_helpdb "mydb"
name  db_size       owner  dbid   created      durability
status
----   ------------ -----  ----   ------------ ----------
---------------
mydb          3.0 MB    sa     5  July 09, 2010       full
select into/bulkcopy/pllsort, trunc log on chkpt, async log service

device_fragments                 size         usage
created                    free kbytes
-----------------------------  -----------  --------------------
------------------------- ----------------
master                     2.0 MB                      data only
Jul  2 2010  1:59PM                     320
log_disk                   1.0 MB                      log only
Jul  2 2010  1:59PM         not applicable

--------------------------------------------------------------
```

```
log only free kbytes = 1018
device      segment
--------    ----------
log_disk    logsegment
master      default
master       system
```

Understanding the user log cache (ULC) architecture

Adaptive Server's logging architecture features the user log cache, or ULC, by which each task owns its own log cache. No other task can write to this cache, and the task continues writing to the user log cache whenever a transaction generates a log record. When the transaction commits or aborts, or the user log cache is full, the user log cache is flushed to the common log cache, shared by all the current tasks, which is then written to the disk.

Flushing the ULC is the first part of a commit or abort operation. It requires the following steps, each of which can cause delay or increase contention:

1. Obtaining a lock on the last log page.
2. Allocating new log pages if necessary.
3. Copying the log records from the ULC to the log cache.

   The processes in steps 2 and 3 require a lock to be held on the last log page, which prevents any other tasks from writing to the log cache or performing commit or abort operations.
4. Flushing the log cache to disk.

   Step 4 requires repeated scanning of the log cache to issue write commands on dirty buffers.

   Repeated scanning can cause contention on the buffer cache spinlock to which the log is bound. Under a large transaction load, contention on this spinlock can be significant.

When to use ALS

You can enable ALS on any database that has at least one of the following performance issues, as long as your systems runs four or more online engines:

- Heavy contention on the last log page – sp_sysmon output in the Task Management Report section shows a significantly high value. This example shows a log page under contention:

```
Task Management            per sec    per xact     count   % of total
----------------------   ----------  ----------  --------     --------
Log Semaphore Contention       58.0         0.3     34801         73.1
```

- Heavy contention on the cache manager spinlock for the log cache – sp_sysmon output in the Data Cache Management Report section for the database transaction log cache shows a high value in the Spinlock Contention section. For example:

```
Task Management            per sec    per xact     count   % of total
----------------------   ----------  ----------  --------     --------
Spinlock Contention             n/a         n/a       n/a         40.0
```

- Underutilized bandwidth in the log device.

Note Use ALS only when you identify a single database with high transaction requirements, since setting ALS for multiple databases may cause unexpected variations in throughput and response times. If you want to configure ALS on multiple databases, first check that your throughput and response times are satisfactory.

Using the ALS

Two threads scan the dirty buffers (buffers full of data not yet written to the disk), copy the data, and write it to the log. These threads are:

- The user log cache (ULC) flusher
- The log writer.

ULC flusher

The ULC flusher is a system task thread dedicated to flushing the user log cache of a task into the general log cache. When a task is ready to commit, the user enters a commit request into the flusher queue. Each entry has a handle, by which the ULC flusher can access the ULC of the task that queued the request. The ULC flusher task continuously monitors the flusher queue, removing requests from the queue and servicing them by flushing ULC pages into the log cache.

Log writer

When the ULC flusher has finished flushing the ULC pages into the log cache, it queues the task request into a wakeup queue. The log writer patrols the dirty buffer chain in the log cache, issuing a write command if it finds dirty buffers, and monitors the wakeup queue for tasks whose pages are all written to disk. Since the log writer patrols the dirty buffer chain, it knows when a buffer is ready to write to disk.

Housekeeper wash task improves CPU utilization

The housekeeper wash task (which sp_who reports as `HK WASH`) typically runs as a low-priority task, and runs only during idle cycles when Adaptive Server has no user tasks to process. While running, the wash task automatically writes dirty buffers to disk (called free writes) and performs other maintenance tasks. These writes result in improved CPU utilization and a decreased need for buffer washing during transaction processing. They also reduce the number and duration of checkpoint spikes— times when the checkpoint process causes a short, sharp rise in disk writes.

By default, the housekeeper garbage collection operates at the priority level of an ordinary user and cleans up data that was logically deleted and resets the rows so the tables have space again. If Adaptive Server is configured for threaded mode, use sp_bindexeclass or sp_setpsexe to set the housekeeper task to a higher priority level.

See Chapter 11, “Diagnosing System Problems,” in the *System Administration Guide: Volume 1* for more information on the housekeeper tasks and for information about resetting their priorities.

Side effects of the housekeeper wash task

If the housekeeper wash task can flush all active buffer pools in all configured caches, it wakes up the checkpoint task.

The checkpoint task determines whether it can checkpoint the database. If it can, it writes a checkpoint log record indicating that all dirty pages have been written to disk. The additional checkpoints that occur as a result of the housekeeper wash task may improve recovery speed for the database.

In applications that repeatedly update the same database page, the housekeeper wash may initiate some database writes that are not necessary. Although these writes occur only during the server's idle cycles, they may be unacceptable on systems with overloaded disks.

Configuring the housekeeper wash task

System administrators can use the housekeeper free write percent configuration parameter to control the side effects of the housekeeper wash task. This parameter specifies the maximum percentage by which the housekeeper wash task can increase database writes. Valid values are 0 – 100.

By default, housekeeper free write percent is set to 1, which allows the housekeeper wash task to continue to wash buffers as long as database writes do not increase by more than 1 percent. On most systems, work done by the housekeeper wash task at the default setting results in improved performance and recovery speed. Setting housekeeper free write percent too high can degrade performance. If you want to increase the value, increase by only 1 or 2 percent each time.

A dbcc tune option, deviochar, controls the size of batches that the housekeeper can write to disk at one time.

See "Increasing the Housekeeper Batch Limit" in Chapter 2, "Monitoring Performance with sp_sysmon," in *Performance and Tuning Series: Monitoring Adaptive Server with sp_sysmon.*

Changing the percentage by which writes can be increased

Use sp_configure to change the percentage by which database writes can be increased as a result of the housekeeper wash task:

```
sp_configure "housekeeper free write percent", value
```

For example, to stop the housekeeper wash task from working when the frequency of database writes reaches 2 percent above normal enter:

```
sp_configure "housekeeper free write percent", 2
```

Disabling the housekeeper wash task

Disable the housekeeper wash task to establish a more controlled environment in which primarily user tasks are running. To disable the housekeeper wash task, set the value of the housekeeper free write percent parameter to 0:

```
sp_configure "housekeeper free write percent", 0
```

There is no configuration parameter to shut down the housekeeper chores task, although you can set sp_setpsexe to lower its priority.

Allowing the housekeeper wash task to work continuously

To allow the housekeeper wash task to work whenever there are idle CPU cycles, regardless of the percentage of additional database writes, set the value of the housekeeper free write percent parameter to 100:

```
sp_configure "housekeeper free write percent", 100
```

See *Performance and Tuning Series: Monitoring Adaptive Server with sp_sysmon.*

Measuring CPU usage

This section describes how to measure CPU usage on machines with a single processor and on those with multiple processors.

Single-CPU machines

There is no correspondence between your operating system's reports on CPU usage and the Adaptive Server internal "CPU busy" information.

A multithreaded database engine in process mode is not allowed to block on I/O. While asynchronous disk I/O is being performed, Adaptive Server services other user tasks that are waiting to be processed. If there are no tasks to perform, Adaptive Server enters a busy-wait loop, waiting for completion of the asynchronous disk I/O. This low-priority busy-wait loop can result in high CPU usage, but because of its low priority, it is generally harmless.

Adaptive Serves in threaded mode can block on I/O.

Note In process mode, it is normal for an Adaptive Server to exhibit high CPU usage while performing an I/O-bound task.

Using *sp_monitor* to measure CPU usage

Use sp_monitor to see the percentage of time Adaptive Server uses the CPU during an elapsed time interval:

```
last_run                    current_run               seconds
-------------------------   ------------------------  ----------
       Jul 25 2009 5:25PM         Jul 28 2009 5:31PM         360

cpu_busy                 io_busy                 idle
------------------------ ----------------------- -----------------
5531(359)-99%                           0(0)-0%       178302(0)-0%

packets_received         packets_sent            packet_errors
------------------------ ----------------------- ------------------
57650(3599)                         60893(7252)               0(0)

total_read         total_write       total_errors     connections
------------------ ----------------- ---------------- --------------
190284(14095)             160023(6396)             0(0)           178(1)
```

For more information about sp_monitor, see the *Adaptive Server Enterprise Reference Manual.*

Using *sp_sysmon* to measure CPU usage

sp_sysmon provides more detailed information than sp_monitor. The “Kernel Utilization” section of the sp_sysmon report displays how busy the engine was during the sample run. The percentage in this output is based on the time that CPU was allocated to Adaptive Server; it is not a percentage of the total sample interval.

The `CPU Yields by Engine` section displays information about how often the engine yielded to the operating system during the interval.

See *Performance and Tuning Series: Monitoring Adaptive Server with sp_sysmon*.

Operating system commands and CPU usage

Operating system commands for displaying CPU usage are documented in the Adaptive Server installation and configuration guides.

If your operating system tools show that CPU usage is more than 85 percent most of the time, consider using a multi-CPU environment or offloading some work to another Adaptive Server.

Determining when to configure additional engines

When you are determining whether to add additional engines, consider:

- Load on existing engines
- Contention for resources, such as locks on tables, disks, and cache spinlocks
- Response time

If the load on existing engines is more than 80 percent, adding an engine should improve response time, unless contention for resources is high or the additional engine causes contention.

Before configuring more engines, use sp_sysmon to establish a baseline. Look at the sp_sysmon output for the following sections in Monitoring Performance with sp_sysmon in *Performance and Tuning Series: Monitoring Adaptive Server with sp_sysmon*. In particular, study the lines or sections in the output that may reveal points of contention:

- Logical Lock Contention
- Address Lock Contention
- ULC Semaphore Requests
- Log Semaphore Requests
- Page Splits
- Lock Summary

- Spinlock Contention
- I/Os Delayed by

After increasing the number of engines, run sp_sysmon again under similar load conditions, and check the “Engine Busy Utilization” section in the report and the possible points of contention listed above.

Taking engines offline

If you are running Adaptive Server in process mode, use sp_engine to take engines online and offline. Adaptive Server does not accept sp_engine in threaded mode. See the sp_engine entry in the *Adaptive Server Enterprise Reference Manual: Procedures*.

Enabling engine-to-CPU affinity

By default, there is no affinity between CPUs and engines in Adaptive Server. You may see slight performance gains in high-throughput environments by establishing affinity of engines to CPUs.

Not all operating systems support CPU affinity; on such systems, the dbcc tune command is silently ignored. You must reissue dbcc tune each time Adaptive Server is restarted. Each time CPU affinity is turned on or off, Adaptive Server prints a message in the error log indicating the engine and CPU numbers affected:

```
Engine 1, cpu affinity set to cpu 4.
Engine 1, cpu affinity removed.
```

The syntax is:

dbcc tune(cpuaffinity, *start_cpu* [, on | off])

start_cpu specifies the CPU to which engine 0 is to be bound. Engine 1 is bound to the CPU numbered (*start_cpu* + 1). The formula for determining the binding for engine *n* is:

```
((start_cpu + n) % number_of_cpus
```

Valid CPU numbers are 0 – the number of CPUs minus 1.

On a four-CPU machine (with CPUs numbered 0 – 3) and a four-engine Adaptive Server, this command:

```
dbcc tune(cpuaffinity, 2, "on")
```

Gives this result:

Engine	CPU	
0	2	(the *start_cpu* number specified)
1	3	
2	0	
3	1	

On the same machine, with a three-engine Adaptive Server, the same command causes the following affinity:

Engine	CPU
0	2
1	3
2	0

CPU 1 is not used by Adaptive Server.

To disable CPU affinity, use -1 instead of *start_cpu*, and specify off for the setting:

```
dbcc tune(cpuaffinity, -1, "off")
```

Enable CPU affinity without changing the value of *start_cpu* by using -1 and on for the setting:

```
dbcc tune(cpuaffinity, -1, "on")
```

The default value for *start_cpu* is 1 if CPU affinity has not been previously set.

To specify a new value of *start_cpu* without changing the on/off setting, use:

```
dbcc tune (cpuaffinity, start_cpu)
```

If CPU affinity is currently enabled, and the new *start_cpu* differs from its previous value, Adaptive Server changes the affinity for each engine.

If CPU affinity is off, Adaptive Server notes the new *start_cpu* value, and the new affinity takes effect the next time CPU affinity is turned on.

To see the current value and whether affinity is enabled, use:

```
dbcc tune(cpuaffinity, -1)
```

This command prints only current settings to the error log and does not change the affinity or the settings.

Multiprocessor application design guidelines

If you are moving applications from a single-CPU environment to an SMP environment, this section discusses some issues to consider.

Increased throughput on multiprocessor Adaptive Servers makes it more likely that multiple processes may try to access a data page simultaneously. Adhere to the principles of good database design to avoid contention. These are some of the application design considerations that are especially important in an SMP environment.

- Multiple indexes – the increased throughput of SMP may result in increased lock contention when allpages-locked tables with multiple indexes are updated. Allow no more than two or three indexes on any table that is updated often.

  For information about the effects of index maintenance on performance, see *Performance and Tuning Series: Monitoring Adaptive Server with sp_sysmon*.

- Managing disks – the additional processing power of SMP may increase demands on the disks. Spread data across multiple devices for heavily used databases.

  See *Performance and Tuning Series: Monitoring Adaptive Server with sp_sysmon.*

- Adjusting the fillfactor for create index commands – because of the added throughput with multiple processors, setting a lower fillfactor may temporarily reduce contention for the data and index pages.

- Transaction length – transactions that include many statements or take a long time to run may result in increased lock contention. Keep transactions as short as possible, and avoid holding locks—especially exclusive or update locks—while waiting for user interaction. Ensure that the underlying storage provides both adequate bandwidth and sufficiently low latencies.

- Temporary tables – do not cause contention, because they are associated with individual users and are not shared. However, if multiple user processes use tempdb for temporary objects, there may be some contention on the system tables in tempdb. Use multiple temporary databases or Adaptive Server version 15.0.2 and later to alleviate contention on tempdb’s system tables.

  See Chapter 7, “tempdb Performance Issues,” in *Performance and Tuning Series: Physical Database Tuning*.

CHAPTER 4 Distributing Engine Resources

This chapter explains how to assign execution attributes, how Adaptive Server interprets combinations of execution attributes, and how to predict the impact of various execution attribute assignments on the system.

Understanding how Adaptive Server uses CPU resources is a prerequisite for understanding the discussion about distributing engine resources. For more information, see Chapter 3, "Using Engines and CPUs."

Successfully distributing resources

The interactions among execution objects in an Adaptive Server environment are complex. Furthermore, every environment is different: each involves its own mix of client applications, logins, and stored procedures, and is characterized by the interdependencies between these entities.

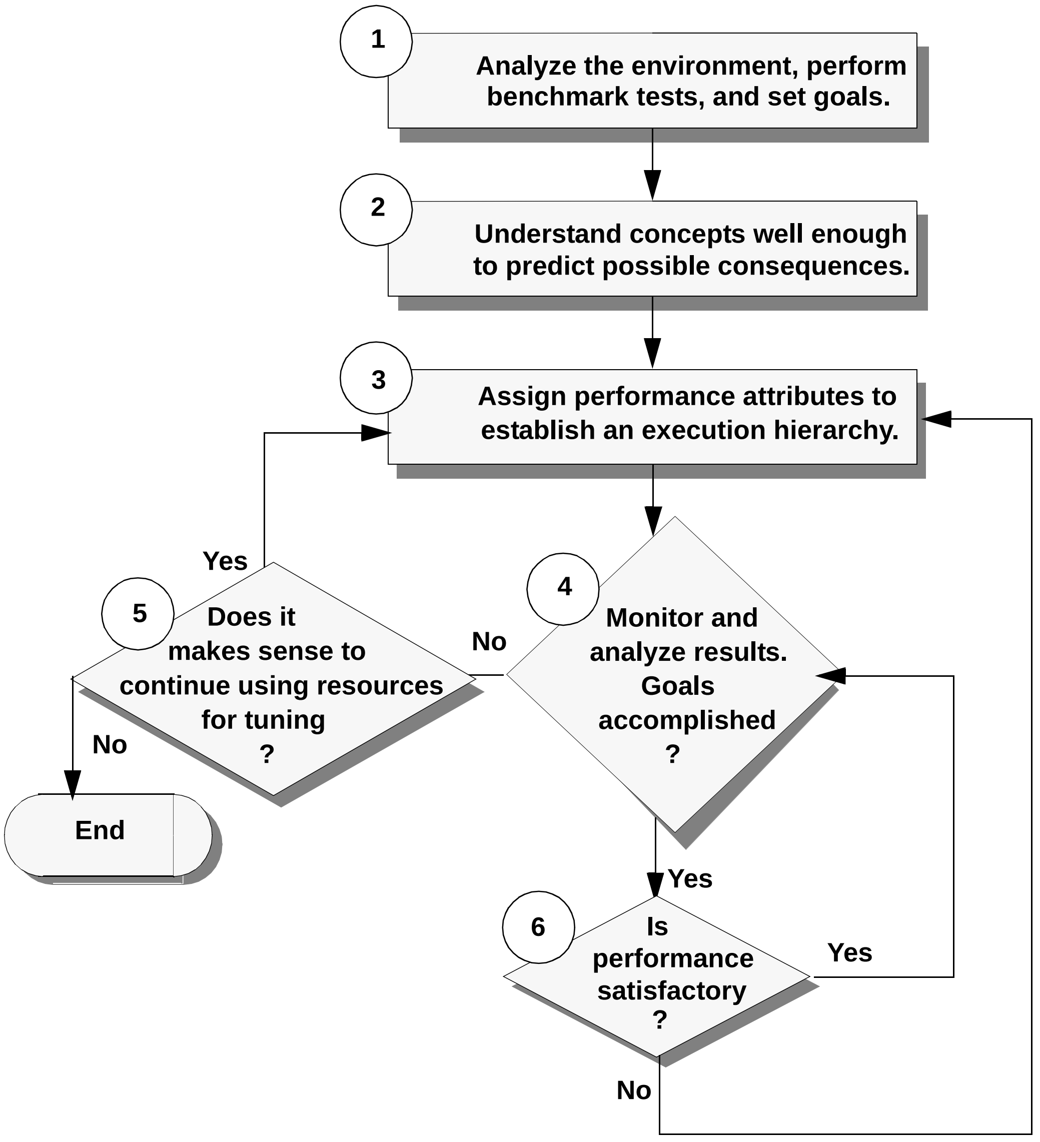
Implementing execution precedence without having studied the environment and the possible implications can lead to unexpected (and negative) results.

For example, say you have identified a critical execution object and you want to raise its execution attributes to improve performance, either permanently or on a per-session basis. If the execution object accesses the same set of tables as one or more other execution objects, raising its execution priority can lead to performance degradation due to lock contention among tasks at different priority levels.

Because of the unique nature of every Adaptive Server environment, Sybase cannot provide a detailed procedure for assigning execution precedence that makes sense for all systems. However, this section provides guidelines, procedures to try, and a discussion of common issues.

Figure 4-1 shows the steps involved in assigning execution attributes.

Figure 4-1: Process for assigning execution precedence


1
Analyze the environment, perform benchmark tests, and set goals.
2
Understand concepts well enough to predict possible consequences.
3
Assign performance attributes to establish an execution hierarchy.
4
Monitor and analyze results. Goals accomplished ?
No
5
Does it makes sense to continue using resources for tuning ?
Yes
No
End
Yes
6
Is performance satisfactory ?
Yes
No

1 Study the Adaptive Server environment:

- Analyze the behavior of all execution objects and categorize them as much as possible.

- Understand interdependencies and interactions between execution objects.
- Perform benchmark tests to use as a baseline for comparison after establishing precedence.
- Think about how to distribute processing in a multiprocessor environment.
- Identify the critical execution objects for which you want to enhance performance.
- Identify the noncritical execution objects that can afford decreased performance.
- Establish a set of quantifiable performance goals for the execution objects identified in the last two items.

  See “Environment analysis and planning” on page 59.

2. Understand the effects of using execution classes:

   - Understand the basic concepts associated with execution class assignments.
   - Decide whether to create one or more user defined-execution classes.
   - Understand the implications of different class level assignments—how do assignments affect the environment in terms of performance gains, losses, and interdependencies?

   See n.

3. Assign execution classes and any independent engine affinity attributes.
4. After making execution precedence assignments, analyze the running Adaptive Server environment:

   - Run the benchmark tests you used in step 1 and compare the results.
   - If the results are not what you expect, take a closer look at the interactions between execution objects, as outlined in step 1.
   - Investigate dependencies that you might have missed.

   See “Results analysis and tuning” on page 61.

5. Fine-tune the results by repeating steps 3 and 4 as many times as necessary.
6. Monitor the environment over time.

Environment analysis and planning

Environment analysis and planning involves:

- Analyzing the environment
- Performing benchmark tests to use as a baseline
- Setting performance goals

Analyzing the environment

Study and understand how Adaptive Server objects interact with your environment so that you can make decisions about how to achieve the performance goals you set.

Analysis involves these two phases:

- Phase 1 – analyze the behavior of each execution object.
- Phase 2 – use the results from the object analysis to make predictions about interactions between execution objects within the Adaptive Server system.

First, make a list containing every execution object that can run in the environment. Then, classify each execution object and its characteristics. Categorize the execution objects with respect to each other in terms of importance. For each, decide which of the following applies:

- It is a highly critical execution object needing enhanced response time,
- It is an execution object of medium importance, or
- It is a noncritical execution object that can afford slower response time.

Phase 1 – execution object behavior

Typical classifications include intrusive/unintrusive, I/O-intensive, and CPU-intensive. For example, identify each object as intrusive or unintrusive, I/O intensive or not, and CPU intensive or not. You will probably need to identify additional issues specific to the environment to gain useful insight.

Two or more execution objects running on the same Adaptive Server are intrusive when they use or access a common set of resources.

Intrusive applications	
Effect of assigning attributes	Assigning high-execution attributes to intrusive applications might degrade performance.
Example	Consider a situation in which a noncritical application is ready to release a resource, but becomes blocked when a highly-critical application starts executing. If a second critical application needs to use the blocked resource, then execution of this second critical application is also blocked

If the applications in the Adaptive Server environment use different resources, they are unintrusive.

Unintrusive applications	
Effect of assigning attributes	You can expect enhanced performance when you assign preferred execution attributes to an unintrusive application.
Example	Simultaneous distinct operations on tables in different databases are unintrusive. Two operations are also unintrusive if one is compute bound and the other is I/O bound.

I/O-intensive and CPU-intensive execution objects

When an execution object is I/O intensive, it might help to give it the EC1 predefined execution class attributes (see “Types of execution classes” on page 63). An object performing I/O does not normally use an entire time period, and yields the CPU before waiting for I/O to complete.

By giving preference to I/O-bound Adaptive Server tasks, Adaptive Server ensures that these tasks are runnable as soon as the I/O is finished. By letting the I/O take place first, the CPU should be able to accommodate both I/O-bound and compute-bound types of applications and logins.

Phase 2 – the entire environment

Follow up on phase 1, in which you identified the behavior of the execution objects, by thinking about how applications interact.

Typically, a single application behaves differently at different times; that is, it might be alternately intrusive and unintrusive, I/O bound, and CPU intensive. This makes it difficult to predict how applications will interact, but you can look for trends.

Organize the results of the analysis so that you understand as much as possible about each execution object with respect to the others. For example, you might create a table that identifies the objects and their behavior trends.

Using Adaptive Server monitoring tools (for example, the monitoring tables) is one of the best ways to understand how execution objects affect the environment. See *Performance and Tuning Series: Monitoring Tables*.

Performing benchmark tests

Perform benchmark tests before assigning any execution attributes so that you have the results to use as a baseline after making adjustments.

Tools that can help you understand system and application behavior include:

- Monitoring tables – provide both a system-wide view or performance and details about objects and users. See *Performance and Tuning Series: Monitoring Tables*.
- sp_sysmon – is a system procedure that monitors system performance for a specified time interval, then prints an ASCII text-based report. See *Performance and Tuning Series: Monitoring Adaptive Server with sp_sysmon*.

Setting goals

Establish a set of quantifiable performance goals. These should be specific numbers based on the benchmark results and your expectations for improving performance. Use these goals to direct you in assigning execution attributes.

Results analysis and tuning

After you configure the execution hierarchy, analyze the running Adaptive Server environment:

1. Run the same benchmark tests you ran before assigning the execution attributes, and compare the results to the baseline results.
2. Use Adaptive Server Monitor or sp_sysmon to ensure there is good distribution across all the available engines. Check the “Kernel Utilization”. See *Performance and Tuning Series: Monitoring Adaptive Server with sp_sysmon*.

3. If the results are not what you expected, take a closer look at the interactions between execution objects. Look for inappropriate assumptions and dependencies that you might have missed.
4. Make adjustments to performance attributes.
5. Fine-tune the results by repeating these steps as many times as necessary to monitor your environment over time.

Managing preferred access to resources

Most performance tuning techniques give you control at either the system level or at the specific query level. Adaptive Server also gives you control over the relative performance of simultaneously running tasks.

Unless you have superfluous resources, the need for control at the task level is greater in parallel execution environments because there is more competition for limited resources.

Use system procedures to assign execution attributes that indicate which tasks should be given preferred access to resources. The logical process manager uses the execution attributes when it assigns priorities to tasks and tasks to engines.

In effect, assigning execution attributes lets you suggest to Adaptive Server how to distribute engine resources between client applications, logins, and stored procedures in a mixed workload environment.

Each client application or login can initiate many Adaptive Server tasks. In a single-application environment, you can distribute resources at the login and task levels to enhance performance for chosen connections or sessions. In a multiple application environment, you can distribute resources to improve performance for selected applications and for chosen connections or sessions.

Warning! Assign execution attributes with caution.

Arbitrary changes in the execution attributes of one client application, login, or stored procedure can adversely affect the performance of others.

Types of execution classes

An execution class is a specific combination of execution attributes that specify values for task priority and task-to-thread pool affinity (or task-to-engine affinity in process mode). You can bind an execution class to one or more execution objects, which are client applications, logins, service classes, and stored procedures.

There are two types of execution classes—predefined and user-defined. Adaptive Server provides three predefined execution classes:

- EC1 – has the most preferred attributes.
- EC2 – has average values of attributes.
- EC3 – has nonpreferred values of attributes.

Objects associated with EC2 are given average preference for engine resources. If an execution object is associated with EC1, Adaptive Server considers it to be critical and tries to give it preferred access to resources.

Any execution object associated with EC3 is considered to be least critical and does not receive resources until the execution objects associated with EC1 and EC2 are executed. By default, execution objects have EC2 attributes.

To change an execution object's execution class from the EC2 default, use sp_bindexeclass, described in "Assigning execution classes" on page 68.

Create user-defined execution classes by combining the execution attributes in ways that best reflect your site's needs. Reasons for doing this include:

- EC1, EC2, and EC3 do not accommodate all combinations of attributes that might be useful.
- Associating execution objects with a particular group of engines would improve performance.
- Binding service tasks like the housekeeper task, `LICENSE HEARTBEAT`, and so on to their own thread pool

Execution class attributes

Each predefined or user-defined execution class comprises of a combination of three attributes: base priority, timeslice, and thread pool affinity (engine affinity in process mode). These attributes determine performance characteristics during execution.

The attributes for the predefined execution classes, EC1, EC2, and EC3, are fixed, as shown in Table 4-1.

Table 4-1: Fixed-attribute composition of predefined execution classes

Execution class level	**Base priority attribute**	**Time slice attribute**	**Engine affinity attribute**
EC1	High	Time slice > t	None
EC2	Medium	Time slice = t	None
EC3	Low	Time slice < t	Engine with the highest engine ID number

By default, a task on Adaptive Server operates with the same attributes as EC2: its base priority is medium, its time slice is set to one tick, and it can run on any engine.

Base priority

Assign base priority when you create a task. The values are “high,” “medium,” and “low.” There is a run queue for each priority for each engine, and the global run queue also has a queue for each priority.

When a thread pool looks for a task to run, it first checks its own high priority run queue, then the high priority global run queue, then its own medium priority run queue, and so on. The result is, runnable tasks in the high priority run queues are scheduled onto thread pools more quickly than tasks in the other queues.

The scheduler search space refers to where engine schedulers look for work (run queue checking):

- In process mode, the scheduler search space is server-wide, meaning all engines share a global run queue. Engines check the run queues of all other engines.
- In threaded mode, the scheduler search space is specific to the thread pool. Each engine thread pool has its own global queue, and the engines within that pool look for tasks associated only with that pool.

During execution, Adaptive Server can temporarily change a task's priority if necessary. A task's priority can be greater than or equal to, but never lower than, its base priority.

When you create a user-defined execution class, you can assign the values high, medium, or low to the task.

Setting the task priority

Task priority is an attribute of an execution class that is set with sp_bindexeclass. The current_priority column from sp_showpsexe output shows the priority level for the current task execution settings:

```
sp_showpsexe
spid          appl_name                      login_name
                    exec_class                          current_priority
          task_affinity
------        ------------------------------ ------------------------------
                   ------------------------------ ---------------
          ------------------------------
     6                                  NULL                           NULL
                                            NULL                 LOW
          syb_default_pool
     7                                  NULL                           NULL
                                            NULL              MEDIUM
          syb_default_pool
     8                                  NULL                           NULL
                                            NULL                 LOW
          syb_default_pool
    13                                  isql                             sa
                                             EC2              MEDIUM
           syb_default_pool
```

In threaded mode, the `task_affinity` column indicates the name of the thread pool. In process mode, it indicates the name of the engine group.

Use sp_setpsexe to set the priority for a specific task. For example, to set the isql task in the example above to a priority level of HIGH, use:

```
sp_setpsexe 13, 'priority', 'HIGH'
```

When you set task priorities, consider that:

- You set priorities for Adaptive Server tasks, not operating system threads.
- Priorities are relative to other tasks. For example, if a user thread pool contains only tasks from a single execution class, setting the priority of that class has no effect since all tasks are running at the same priority.

Task-to-engine affinity

In a multiengine environment, any available engine can process the next task in the global run queue. The engine affinity attribute lets you assign a task to an engine or to a group of engines (in threaded mode, this is done with thread pools).

To organize task-to-engine affinity:

- Associate less critical execution objects with a defined group of engines to restrict the object to a subset of the total number of engines. This reduces processor availability for those objects. The more critical execution objects can execute on any Adaptive Server engine, so performance for them improves because they have the benefit of the resources that the less critical ones are deprived of.
- Associate more critical execution objects with a defined group of engines to which less critical objects do not have access. This ensures that the critical execution objects have access to a known amount of processing power.
- In process mode, when optimal performance for a network-intensive task is the primary concern, administrators can use task-to-engine affinity coupled with dynamic listeners to ensure that tasks run on the same engine as all the tasks' network I/O. In threaded mode, this is not required due to the lack of dedicated network engines.

EC1 and EC2 do not set engine affinity for the execution object; however, EC3 sets affinity to the Adaptive Server engine with the highest engine number in the current configuration.

Use sp_addengine to create engine groups and sp_addexeclass to bind execution objects to an engine group. If you do not want to assign engine affinity for a user-defined execution class, use *ANYENGINE* as the engine group parameter.

In threaded mode, use create thread pool to create a new thread pool. Use sp_addexeclass to bind execution objects to thread pools.

Note The engine affinity attribute is not used for stored procedures.

Engine group affinity when switching modes

Engine groups do not exist in threaded mode. When you switch from threaded to process mode, execution classes are assigned to the default engine groups. For example, if you switch from threaded to process mode and then add the Eng_Group execution class and associate it with engine number 3, the default execution classes EC1 and EC2 are associated with the ANYENGINE engine group, and EC3, with the highest engine number, is associated with the LASTONLINE engine group:

```
sp_showexeclass
classname                      priority   engine_group
              engines
------------------------------ ---------- ------------------------------
              ------------------------------
EC1                                  HIGH                      ANYENGINE
                                         ALL
EC2                                MEDIUM                      ANYENGINE
                                         ALL
EC3                                   LOW                     LASTONLINE
                                           0
Eng_Group                             LOW               new_engine_group
                                           3
```

When you switch to threaded mode, execution classes lose their engine group affinity and are assigned to syb_default_pool. In threaded mode, the example above becomes:

```
sp_showexeclass
classname                     priority     threadpool
------------------------------ ----------   ------------------------------
EC1                            HIGH         syb_default_pool
EC2                            MEDIUM       syb_default_pool
EC3                            LOW          syb_default_pool
Eng_Group                      LOW          new_engine_group
```

Setting execution class attributes

Implement and manage execution hierarchy for client applications, logins, service tasks, and stored procedures using the categories of system procedures listed in Table 4-2.

Table 4-2: System procedures for managing execution object precedence

Category	Description	System procedures
User-defined execution class	Create and drop a user-defined class with custom attributes or change the attributes of an existing class.	• sp_addexeclass • sp_dropexeclass
Execution class binding	Bind and unbind predefined or user-defined classes to client applications, service tasks, and logins.	• sp_bindexeclass • sp_unbindexeclass
For the session only (“on the fly”)	Set and clear attributes of an active session only.	• sp_setpsexe • sp_clearpsexe
Engines	Add engines to and drop engines from engine groups; create and drop engine groups.	• sp_addengine • sp_dropengine
Reporting	Report on engine group assignments, application bindings, and execution class attributes.	• sp_showcontrolinfo • sp_showexeclass • sp_showpsexe

See the *Reference Manual: Procedures*.

Assigning execution classes

The following example illustrates how to assign preferred access to resources to an execution object by associating it with the EC1 execution class. In this case, the execution object is a combination of application and login.

For example, if you decide the “sa” login must get results from isql as quickly as possible, issue sp_bindexeclass with the preferred execution class, EC1, so Adaptive Server grants execution preference to login “sa” when it executes isql:

```
sp_bindexeclass sa, LG, isql, EC1
```

This statement specifies that whenever a login (LG) called “sa” executes the isql application, the “sa” login task executes with EC1 attributes. Adaptive Server improves response time for the “sa” login by placing it in a high-priority run queue, so it is assigned to an engine more quickly

Scheduling service tasks

Adaptive Server allows you to manage service tasks (housekeeper, checkpoint, Replication Agent threads, and so on) with the sp_bindexeclass 'sv' execution class parameter. Binding individual service tasks to execution classes binds these tasks to thread pools, which lets you control dedicated resources for high-priority tasks, and keep service tasks from competing with user tasks.

Note In process mode, you cannot schedule service tasks.

For example, you can:

- Bind the HK WASH housekeeping task to a specific service task.
- Establish a Replication Agent pool and execution class with one thread per Replication Agent, giving dedicated resources, but simultaneously creating a more generic thread pool named service_pool, granting one thread to other tasks of lesser importance.

The monServiceTask monitoring table includes information about all service tasks that are bound to an execution class. This example shows the HK WASH and NETWORK HANDLER service tasks bound to the SC execution class:

```
task_id    spid        name
description                  execution_class
----------- ----------- ------------------------------
---------------------------- ----------------------
3932190             6                          HK WASH
                         NULL                       SC
4456482            10                  NETWORK HANDLER
                         NULL                       SC
```

Creating user-defined execution class task affinity

The following steps illustrate how to use system procedures to create a thread pool associated with a user-defined execution class and bind that execution class to user sessions. In this example, the server is used by technical support staff, who must respond as quickly as possible to customer needs, and by managers who are usually compiling reports, and can afford slower response time.

To create the user-defined execution class for this example:

1. Create the thread pool named DS_GROUP that governs the task. For example:

   ```
   create thread pool DS_GROUP with thread count = 4
   ```

2. Use sp_addexeclass to create a user-defined execution classes that have names and attributes you choose. For example:

   ```
   sp_addexeclass DS, LOW, 0, DS_GROUP
   ```

3. Use sp_bindexeclass to associate the user-defined execution class with an execution object. For example with three logins:

   ```
   sp_bindexeclass mgr1, LG, NULL, DS
   sp_bindexeclass mgr2, LG, NULL, DS
   sp_bindexeclass mgr3, LG, NULL, DS
   ```

Perform these steps to create a user-defined execution class if Adaptive Server is configured for process mode:

1. Create an engine group called DS_GROUP, consisting of engine 3:

   ```
   sp_addengine 3, DS_GROUP
   ```

   Expand the group so that it also includes engines 4 and 5:

   ```
   sp_addengine 4, DS_GROUP
   sp_addengine 5, DS_GROUP
   ```

2. Create a user-defined execution class called DS with a priority of “low” and associate it with the DS_GROUP engine group.

   ```
   sp_addexeclass DS, LOW, 0, DS_GROUP
   ```

3. Bind the less critical execution objects to the new execution class.

   For example, bind the manager logins, “mgr1,” “mgr2,” and “mgr3,” to the DS execution class using sp_bindexeclass three times:

   ```
   sp_bindexeclass mgr1, LG, NULL, DS
   sp_bindexeclass mgr2, LG, NULL, DS
   sp_bindexeclass mgr3, LG, NULL, DS
   ```

   The second parameter, LG, indicates that the first parameter is a login name. The third parameter, NULL, indicates that the association applies to any application that the login might be running. The fourth parameter, DS, indicates that the login is bound to the DS execution class.

The result of this example is that the technical support group (not bound to an engine group) is given access to more immediate processing resources than the managers.

How execution class bindings affect scheduling

You can use logical process management to increase the priority of specific logins, of specific applications, or of specific logins executing specific applications. This example looks at:

- An order_entry application, an OLTP application critical to taking customer orders.
- A sales_report application that prepares various reports. Some managers run this application with default characteristics, but other managers run the report at lower priority.
- Other users, who are running other applications at default priorities.

Execution class bindings

The following statement binds order_entry with EC1 attributes, giving higher priority to the tasks running it:

```
sp_bindexeclass order_entry, AP, NULL, EC1
```

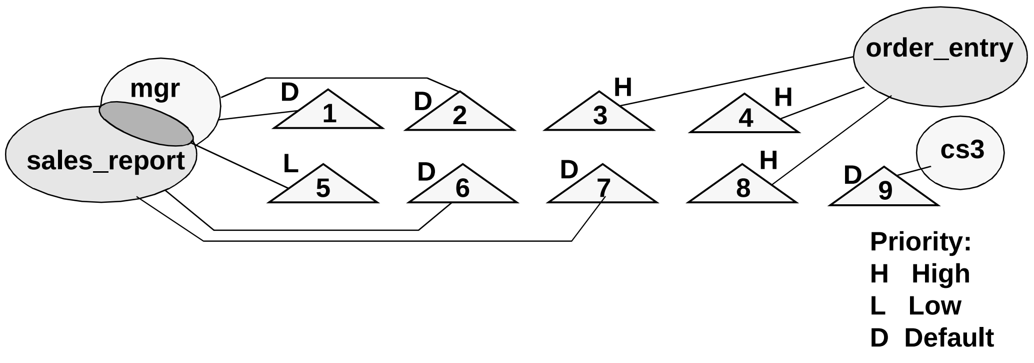
The following sp_bindexeclass statement specifies EC3 when "mgr" runs the sales_reportapplication:

```
sp_bindexeclass mgr, LG, sales_report, EC3
```

This task can execute only when there are no runnable tasks with the EC1 or EC2 attributes.

Figure 4-2 shows four execution objects running tasks. Several users are running the order_entry and sales_report applications. Two other logins are active, "mgr" (logged in once using the sales_report application, and twice using isql) and "cs3" (not using the affected applications).

Figure 4-2: Execution objects and their tasks



When the "mgr" login uses isql (tasks 1 and 2), the task runs with default attributes. But when the "mgr" login uses sales_report, the task runs at EC3. Other managers running sales_report (tasks 6 and 7) run with the default attributes. All tasks running order_entry run at high priority, with EC1 attributes (tasks 3, 4, and 8). "cs3" runs with default attributes.

Engine affinity can affect scheduling in process mode

An engine looking for a task to run first looks in its own high-priority run queues, then in the high-priority global run queue. If there are no high-priority tasks, the engine then checks for medium-priority tasks in its own run queue, then in the medium-priority global run queue, and finally for low-priority tasks.

What happens if a task has affinity to a particular engine? Assume that task 7 in Figure 4-2, a high-priority task in the global run queue, has a user-defined execution class with high priority and affinity to engine number 2, but this engine currently has high-priority tasks queued and is running another task.

If engine 1 has no high-priority tasks queued when it finishes processing task 8 in Figure 4-2, it checks the global run queue, but cannot process task 7 due to the engine binding. Engine 1 then checks its own medium-priority queue, and runs task 15. Although a system administrator assigned the preferred execution class EC1, engine affinity temporarily lowered task 7's execution precedence to below that of a task with EC2.

This effect might be undesirable, or it might be what was intended. You can assign engine affinity and execution classes so that task priority is not what you intended. You can also make assignments so that tasks with low priority might not ever run, or might wait for extremely long times—an important reason to plan and test thoroughly when assigning execution classes and engine affinity.

Note In threaded mode, engines and their assigned tasks exist in completely separate search spaces.

Setting attributes for a session only

Use sp_setpsexe to temporarily change any attribute value temporarily for an active session.

The change in attributes is valid only for the specified spid and is in effect only for the duration of the session, whether it ends naturally or is terminated. Setting attributes using sp_setpsexe neither alters the definition of the execution class for any other process nor does it apply to the next invocation of the active process on which you use it.

To clear attributes set for a session, use sp_clearpsexe.

Getting information about execution classes

Adaptive Server stores the information about execution class assignments in the system tables sysattributes and sysprocesses, and supports several system procedures for determining what assignments have been made.

Use sp_showcontrolinfo to display information about the execution objects bound to execution classes, the Adaptive Server engines in an engine group, and session-level attribute bindings. If you do not specify parameters, sp_showcontrolinfo displays the complete set of bindings and the composition of all engine groups.

sp_showexeclass displays the attribute values of an execution class or all execution classes.

You can also use sp_showpsexe to see the attributes of all running processes.

Determining precedence and scope

Determining the ultimate execution hierarchy between two or more execution objects can be complicated. What happens when a combination of dependent execution objects with various execution attributes makes the execution order unclear?

For example, an EC3 client application can invoke an EC1 stored procedure. Do both execution objects take EC3 attributes, EC1 attributes, or EC2 attributes?

Understanding how Adaptive Server determines execution precedence is important for getting what you want out of your execution class assignments. Two fundamental rules, the precedence rule and the scope rule, can help you determine execution order.

Multiple execution objects and ECs

Adaptive Server uses precedence and scope rules to determine which specification, among multiple conflicting ones, to apply.

Use the rules in this order:

1. Use the precedence rule when the process involves multiple execution object types.
2. Use the scope rule when there are multiple execution class definitions for the same execution object.

Precedence rule

The precedence rule sorts out execution precedence when an execution object belonging to one execution class invokes an execution object of another execution class.

The precedence rule states that the execution class of a stored procedure overrides that of a login, which, in turn, overrides that of a client application.

If a stored procedure has a more preferred execution class than that of the client application process invoking it, the precedence of the client process is temporarily raised to that of the stored procedure for the period of time during which the stored procedure runs. This also applies to nested stored procedures.

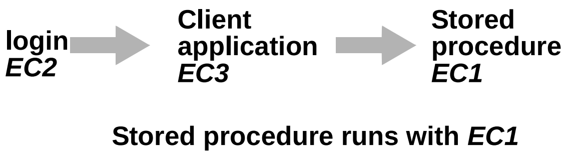
Note Exception to the precedence rule: If an execution object invokes a stored procedure with a less preferred execution class than its own, the execution object's priority is not temporarily lowered.

Precedence rule example

This example illustrates the use of the precedence rule. Suppose there is an EC2 login, an EC3 client application, and an EC1 stored procedure.

The login's attributes override those of the client application, so the login is given preference for processing. If the stored procedure has a higher base priority than the login, the base priority of the Adaptive Server process executing the stored procedure goes up temporarily for the duration of the stored procedure's execution. Figure 4-3 shows how the precedence rule is applied.

Figure 4-3: Use of the precedence rule



What happens when a login with EC2 invokes a client application with EC1 and the client application calls a stored procedure with EC3? The stored procedure executes with the attributes of EC2 because the execution class of a login precedes that of a client application. Using the exception to the precedence rule described in the note above, the priority is not temporarily lowered.

Scope rule

In addition to specifying the execution attributes for an object, you can define its scope when you use sp_bindexeclass *scope*. The object's scope specifies the entities for which the execution class bindings are effective

For example, you can specify that an isql client application run with EC1 attributes, but only when it is executed by an “sa” login. This statement sets the scope of the EC1 binding to the isql client application as the “sa” login (AP indicates an application):

```
sp_bindexeclass isql, AP, sa, EC1
```

Conversely, you can specify that the “sa” login run with EC1 attributes, but only when it executes the isql client application. In this example, the scope of the EC1 binding to the “sa” login is the isql client application:

```
sp_bindexeclass sa, LG, isql, EC1
```

If the scope is set to NULL, the binding is for all interactions.

When a client application has no scope, the execution attributes bound to it apply to any login that invokes the client application.

When a login has no scope, the attributes apply to the login for any process that the login invokes.

The isql parameter in the following command specifies that Transact-SQL applications execute with EC3 attributes for any login that invokes isql, unless the login is bound to a higher execution class:

```
sp_bindexeclass isql, AP, NULL, EC3
```

Combined with the bindings above that grant the “sa” user of isql EC1 execution attributes, and using the precedence rule, an isql request from the “sa” login executes with EC1 attributes. Other processes servicing isql requests from logins that are not “sa” execute with EC3 attributes.

The scope rule states that when a client application, login, service class, or stored procedure is assigned multiple execution class levels, the one with the narrowest scope has precedence. Using the scope rule, you can get the same result if you use:

```
sp_bindexeclass isql, AP, sa, EC1
```

Resolving a precedence conflict

Adaptive Server uses the following rules to resolve conflicting precedence when multiple execution objects and execution classes have the same scope.

- Execution objects not bound to a specific execution class are assigned these default values:

Entity type	Attribute name	Default value
Client application	Execution class	EC2
Login	Execution class	EC2
Stored procedure	Execution class	EC2

- An execution object for which an execution class is assigned has higher precedence than defaults. (An assigned EC3 has precedence over an unassigned EC2).
- If a client application and a login have different execution classes, the login has higher execution precedence than the client application (from the precedence rule).
- If a stored procedure and a client application or login have different execution classes, Adaptive Server uses the one with the higher execution class to derive the precedence when it executes the stored procedure (from the precedence rule).
- If there are multiple definitions for the same execution object, the one with a narrower scope has the highest priority (from the scope rule). For example, the first statement gives precedence to the “sa” login running isql over “sa” logins running any other task:

```
sp_bindexeclass sa, LG, isql, EC1
sp_bindexeclass sa, LG, NULL, EC2
```

Examples: determining precedence

Each row in Table 4-3 contains a combination of execution objects and their conflicting execution attributes.

The “Execution class attributes” columns show execution class values assigned to a process application “AP” belonging to login “LG.”

The remaining columns show how Adaptive Server resolves precedence.

Table 4-3: Conflicting attribute values and Adaptive Server assigned values

Execution class attributes			Adaptive Server-assigned values		
Application (AP)	Login (LG)	Stored procedure (sp_ec)	Application	Login base priority	Stored procedure base priority
EC1	EC2	EC1 (EC3)	EC2	Medium	High (Medium)

Execution class attributes			**Adaptive Server-assigned values**		
Application (AP)	**Login (LG)**	**Stored procedure (sp_ec)**	**Application**	**Login base priority**	**Stored procedure base priority**
EC1	EC3	EC1 (EC2)	EC3	Low	High (Medium)
EC2	EC1	EC2 (EC3)	EC1	High	High (High)
EC2	EC3	EC1 (EC2)	EC3	Low	High (Medium)
EC3	EC1	EC2 (EC3)	EC1	High	High (High)
EC3	EC2	EC1 (EC3)	EC2	Medium	High (Medium)

To test your understanding of the rules of precedence and scope, cover the "Adaptive Server-assigned values" columns in Table 4-3, and predict the values in those columns. To help get you started, this is a description of the scenario in the first row:

- Column 1 – client application, AP, is specified as EC1.
- Column 2 – login, "LG", is specified as EC2.
- Column 3 – stored procedure, sp_ec, is specified as EC1.

At runtime:

- Column 4 – task belonging LG, executing the client application AP, uses EC2 attributes because the class for a login precedes that of an application (precedence rule).
- Column 5 – value of column 5 implies a medium base priority for the login.
- Column 6 – execution priority of the stored procedure sp_ec is raised to high from medium (because it is EC1).

  If the stored procedure is assigned EC3 (as shown in parentheses in column 3), then the execution priority of the stored procedure is medium (as shown in parentheses in column 6) because Adaptive Server uses the highest execution priority of the client application or login and stored procedure.

Example scenario using precedence rules

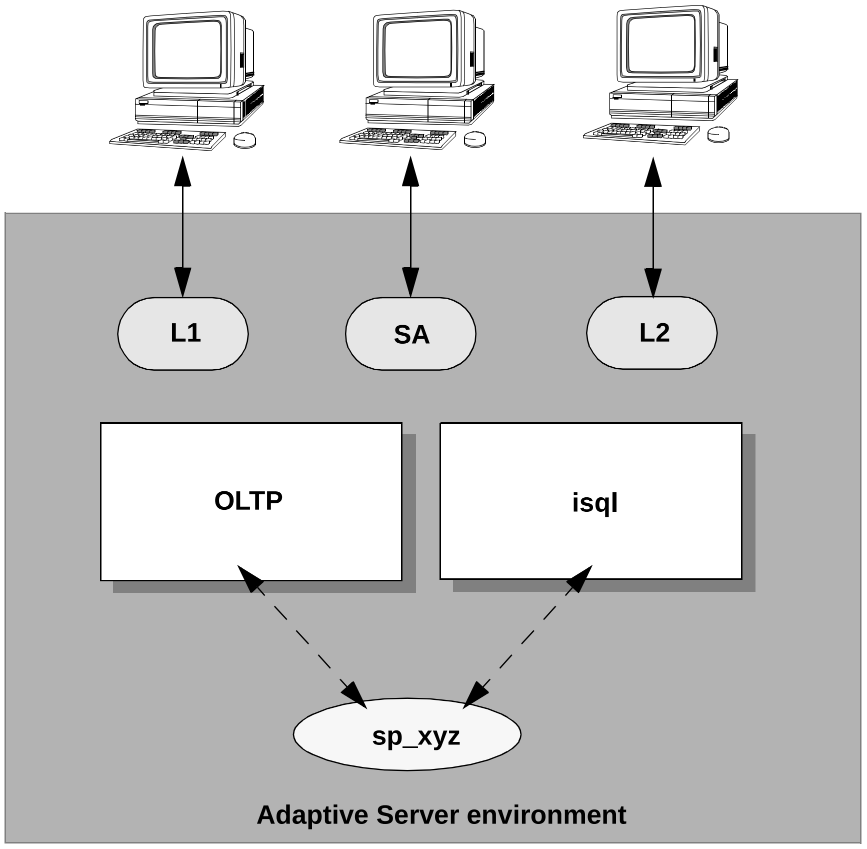
This section presents an example that illustrates how the system administrator interprets execution class attributes, including:

- Planning – the system administrator analyzes the environment, performs benchmark tests, sets goals, and understands the concepts well enough to predict consequences.
- Configuration – the system administrator runs sp_bindexeclass with parameters based on the information gathered in the Planning section.
- Execution characteristics – applications connect with Adaptive Server, using the configuration the system administrator has created.

Figure 4-4 shows two client applications, OLTP and isql, and three Adaptive Server logins, "L1", "sa", and "L2".

sp_xyz is a stored procedure that both the OLTP application and the isql application need to execute.

Figure 4-4: Conflict resolution



Planning

The system administrator performs the analysis described in steps 1 and 2 in "Successfully distributing resources" on page 55 and decides on this hierarchy plan:

- The OLTP application is an EC1 application and the isql application is an EC3 application.
- Login "L1" can run different client applications at different times and has no special performance requirements.

- Login "L2" is a less critical user and should always run with low performance characteristics.
- Login "sa" must always run as a critical user.
- Stored procedure sp_xyz should always run with high performance characteristics. Because the isql client application can execute the stored procedure, giving sp_xyz a high-performance characteristics is an attempt to avoid a bottleneck in the path of the OLTP client application.

Table 4-1 summarizes the analysis and specifies the execution class to be assigned by the system administrator. The tuning granularity gets finer as you descend the table. Applications have the greatest granularity, or the largest scope. The stored procedure has the finest granularity, or the narrowest scope.

Table 4-4: Example analysis of an Adaptive Server environment

Identifier	Interactions and comments	Execution class
OLTP	• Same tables as isql • Highly critical	EC1
isql	• Same tables as OLTP • Low priority	EC3
L1	• No priority assignment	None
sa	• Highly critical	EC1
L2	• Not critical	EC3
sp_xyz	• Avoid "hot spots"	EC1

Configuration

The system administrator executes the following system procedures to assign execution classes (step 3 on page 56):

```
sp_bindexeclass OLTP, AP, NULL, EC1
sp_bindexeclass ISQL, AP, NULL, EC3
sp_bindexeclass sa, LG, NULL, EC1
sp_bindexeclass L2, LG, NULL, EC3
sp_bindexeclass SP_XYZ, PR, sp_owner, EC1
```

Execution characteristics

Following is a series of events that could take place in an Adaptive Server environment with the configuration described in this example:

1. A client logs in to Adaptive Server as "L1" using OLTP.
   - Adaptive Server determines that OLTP is EC1.
   - "L1"does not have an execution class. However, because "L1" logs in to the OLTP application, Adaptive Server assigns the execution class EC1.
   - "L1" executes the stored procedure at a high priority since the object has been assigned execution class EC1.
2. A client logs in to Adaptive Server as "L1" using isql.
   - Because isql is EC3, and "L1" is not bound to an execution class, "L1"executes with EC3 characteristics. This means it runs at low priority and has affinity with the highest numbered engine (as long as there are multiple engines).
   - When "L1"executes sp_xyz, its priority is raised to high because the stored procedure is EC1.
3. A client logs in to Adaptive Server as "sa" using isql.
   - Adaptive Server determines the execution classes for both isql and "sa", using the precedence rule. Adaptive Server runs the system administrator's instance of isql with EC1 attributes. When the system administrator executes sp_xyz, the priority does not change.
4. A client logs in to Adaptive Server as "L2" using isql.
   - Because both the application and login are EC3, there is no conflict. "L2" executes sp_xyz at high priority.

Considerations for engine resource distribution

Making execution class assignments indiscriminately does not usually yield what you expect. Certain conditions yield better performance for each execution object type. Table 4-5 indicates when assigning an execution precedence might be advantageous for each type of execution object.

Table 4-5: When assigning execution precedence is useful

Execution object	Description
Client application	There is little contention for non-CPU resources among client applications.
Adaptive Server login	One login should have priority over other logins for CPU resources.

Execution object	Description
Stored procedure	There are well-defined stored procedure “hot spots.”

It is more effective to lower the execution class of less-critical execution objects than to raise the execution class of a highly critical execution object.

Client applications: OLTP and DSS

Assigning higher execution preference to client applications can be particularly useful when there is little contention for non-CPU resources among client applications.

For example, if an OLTP application and a DSS application execute concurrently, you might be willing to sacrifice DSS application performance if that results in faster execution for the OLTP application. You can assign non-preferred execution attributes to the DSS application so that it gets CPU time only after OLTP tasks are executed.

Unintrusive client applications

Inter-application lock contention is not a problem for an unintrusive application that uses or accesses tables that are not used by any other applications on the system.

Assigning a preferred execution class to such an application ensures that whenever there is a runnable task from this application, it is first in the queue for CPU time.

I/O-bound client applications

If a highly-critical application is I/O bound and the other applications are compute-bound, the compute bound process can use the CPU for full timeslice if it is not blocked for some other reason.

An I/O-bound process, however, yields the CPU each time it performs an I/O operation. Assigning a unpreferred execution class to the compute-bound application enables Adaptive Server to run the I/O-bound process sooner.

Critical applications

If there are one or two critical execution objects among several noncritical ones, try setting affinity to a specific thread pool for the less critical applications. This can result in better throughput for the critical applications.

Adaptive Server logins: high-priority users

If you assign preferred execution attributes to a critical user and maintain default attributes for other users, Adaptive Server does what it can to execute all tasks associated with the high-priority user first.

In process mode, one result of scheduling is that when an engine does not find a task in its local run or a global run queue, it attempts to steal a task from another engine's local run queue. Engines can steal only tasks that have a normal priority, and can never steal a high-priority task for high-priority users. If engine loads are not well-balanced, and the engines running high-priority tasks are heavily loaded, the task-stealing can lead to high-priority tasks being starved of CPU, which is opposite of the intended affect of scheduling, but a natural side effect.

Stored procedures: "hot spots"

Performance issues associated with stored procedures arise when a stored procedure is heavily used by one or more applications. When this happens, the stored procedure is characterized as a hot spot in the path of an application.

Usually, the execution priority of the applications executing the stored procedure is in the medium to low range, so assigning more preferred execution attributes to the stored procedure might improve performance for the application that calls it.

CHAPTER 5 Memory Use and Performance

This chapter describes how Adaptive Server uses the data and procedure caches and other issues affected by memory configuration. In general, the more memory available, the faster Adaptive Server's response time.

Chapter 3, "Configuring Memory," in *System Administration Guide: Volume 2* describes how to determine the best memory configuration values for Adaptive Server, and the memory needs of other server configuration options.

How memory affects performance

Having ample memory reduces disk I/O, which improves performance, since memory access is much faster than disk access. When a user issues a query, data and index pages must be in memory, or read into memory, to examine the values on them. If the pages already reside in memory, Adaptive Server does not need to perform disk I/O.

Adding more memory is inexpensive and easy, but developing around memory problems is expensive. Give Adaptive Server as much memory as possible.

Memory conditions that can cause poor performance include:

- Total data cache size is too small.
- Procedure cache size is too small.
- Only the default cache is configured on an SMP system with several active CPUs, leading to contention for the data cache.
- User-configured data cache sizes are inappropriate for specific user applications.
- Configured I/O sizes are inappropriate for specific queries.
- Audit queue size is inappropriate if auditing feature is installed.

How much memory to configure

Memory is the most important consideration when you are configuring Adaptive Server. Memory is consumed by various configuration parameters, thread pools, procedure caches, and data caches. Correctly setting the values of the configuration parameters and the caches is critical to good system performance.

The total memory allocated during start up is the sum of the memory required for all Adaptive Server configuration requirements. This value is accumulated by Adaptive Server from the read-only configuration parameter total logical memory. The configuration parameter max memory must be greater than or equal to total logical memory. max memory indicates the amount of memory you allow for Adaptive Server needs.

Adaptive Server allocates memory based on the value of total logical memory at start up. However, if you have set the configuration parameter allocate max shared memory, the amount of memory Adaptive Server allocates is based on the value of max memory. This allows a system administrator to tell Adaptive Server to allocate, at start up, the maximum allowed, which may be considerably more than the value of total logical memory at that time.

The key points for memory configuration are:

- The system administrator should determine the size of shared memory available to Adaptive Server and set max memory to this value.

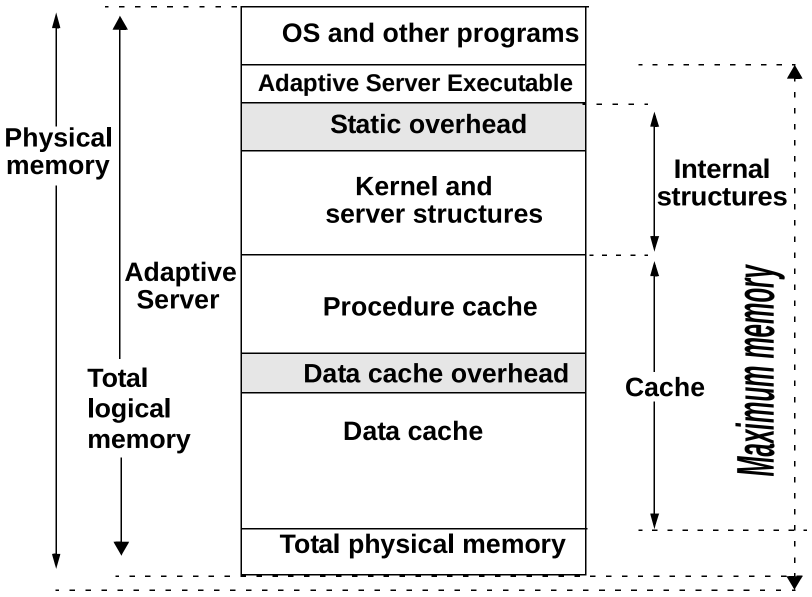
- Set the value for the allocate max shared memory at startup configuration parameter to the fewest number of shared memory segments. This may improve performance, because operating with a large number of shared memory segments may cause performance degradation on certain platforms. See your operating system documentation to determine the optimal number of shared memory segments. Once a shared memory segment is allocated, it cannot be released until the next time you start Adaptive Server.
- The amount of memory available for a new thread pool is determined by the amount of free memory available from max memory. If Adaptive Server has insufficient memory to create the thread pool, it displays an error message indicating the amount you must raise max memory before creating the thread pool. In this example
- If the defaults are insufficient, reconfigure the configuration parameters.
- The difference between max memory and total logical memory is additional memory available for procedure, for data caches, thread pools, or for other configuration parameters.

  The amount of memory to be allocated by Adaptive Server during boot-time is determined by either total logical memory or max memory. If this value is too high:

  - Adaptive Server may not start if the physical resources on your machine are insufficient.
  - If Adaptive Server starts, the operating system page fault rates may rise significantly and you may need to reconfigure the operating system to compensate.

What remains after all other memory needs have been met is available for the procedure cache and the data cache. Figure 5-1 shows how memory is divided.

Figure 5-1: How Adaptive Server uses memory



Dynamic reconfiguration

Adaptive Server allows you to allocate total physical memory dynamically. Many of the configuration parameters that consume memory are dynamic, which means you do not need to restart the server for them to take effect. For example, number of user connections, number of worker processes, and time slice can all be changed dynamically. See Chapter 5: “Setting Configuration Parameters,” in *System Administration Guide: Volume 1* for a complete discussion of configuration parameters, including information on which are dynamic and which are static.

How memory is allocated

Prior to version 12.5 of Adaptive Server, the size of the procedure cache was based on a percentage of the available memory. After you configured the data cache, whatever was left over was allocated to the procedure cache. For Adaptive Server 12.5 and later, the data cache and the procedure cache are specified as absolute values. The sizes of the caches do not change until you reconfigure them.

Use the configuration parameter max memory to establish a maximum setting, beyond which you cannot configure Adaptive Server's total physical memory.

Large allocation in Adaptive Server

Adaptive Server automatically tunes the size of procedure cache allocations to optimize memory use and reduce external fragmentation. When serving repeated internal requests for memory, Adaptive Server initially allocates 2K chunks, then scales up the allocation size to a maximum of 16K chunks, based on past allocation history. This optimization is transparent to the end user, except as it contributes to improved performance.

Caches in Adaptive Server

Adaptive Server includes the procedure and data cache:

- The **procedure cache** is used for stored procedures and triggers and for short-term memory needs such as statistics and query plans for parallel queries.

  Set the procedure cache size to an absolute value using sp_configure, "procedure cache size". See Chapter 5, "Setting Configuration Parameters," in the *System Administration Guide: Volume 1*.

- The **data cache** is used for data, index, and log pages. The data cache can be divided into separate, named caches, with specific databases, or database objects bound to specific caches.

Once the procedure cache and the data cache are configured, there is no division of leftover memory.

Cache sizes and buffer pools

Adaptive Server uses different page sizes for cache and buffer pools:

- Memory pages – (max memory, total logical memory, and so on) are multiples of 2K
- Procedure cache – configured in 2K pages
- Buffer cache – expressed in units of logical page size
- Large I/O – scaled in terms of extents (each extent is 8 pages). For example, if Adaptive Server is configured for an 8K logical page size, large I/O uses a read or write that is 64K.

If you start Adaptive Server and the caches are defined with buffer pools that are not valid for the current logical page size, all memory for such inapplicable buffer pools is reallocated when configuring caches to the default buffer pool in each named cache.

Be careful in how you set up logical page sizes and what you allow for in the buffer pool sizes.

Logical page size	Possible buffer pool sizes
2K	2K, 4K, 16K
4K	4K, 8K, 16K, 32K
8K	8K, 16K, 32K, 64K
16K	16K, 32K, 64K, 128K

Procedure cache

Adaptive Server maintains an MRU/LRU (most recently used/least recently used) chain of stored procedure query plans. As users execute stored procedures, Adaptive Server looks in the procedure cache for a query plan to use. If a query plan is available, it is placed on the MRU end of the chain, and execution begins.

If no plan is in memory, or if all copies are in use, the query tree for the procedure is read from the sysprocedures table. The query tree is then optimized, using the parameters provided to the procedure, and placed at the MRU end of the chain, and execution begins. Plans at the LRU end of the page chain that are not in use are aged out of the cache.

The memory allocated for the procedure cache holds the optimized query plans (and occasionally trees) for all batches, including any triggers.

If more than one user uses a procedure or trigger simultaneously, there will be multiple copies of it in cache. If the procedure cache is too small, a user trying to execute stored procedures or queries that fire triggers receives an error message and must resubmit the query. Space becomes available when unused plans age out of the cache.

Adaptive Server uses the default procedure cache size (in memory pages) at start up. The optimum value for the procedure cache varies from application to application, and it may also vary as usage patterns change. Use procedure cache size to determine the current size of the procedure cache (see Chapter 5, "Setting Configuration Parameters," in *System Administration Guide: Volume 1*).

Getting information about the procedure cache size

When you start Adaptive Server, the error log states how much procedure cache is available.

- proc buffers represents the maximum number of compiled procedural objects that can simultaneously reside in the procedure cache.
- proc headers represents the number of pages dedicated to the procedure cache. Each object in cache requires at least one page.

Monitoring procedure cache performance

sp_sysmon reports on stored procedure executions and the number of times that stored procedures must be read from disk.

See *Performance and Tuning Series: Monitoring Adaptive Server with sp_sysmon*.

If there is not enough memory to load another query tree or plan, or if the maximum number of compiled objects is already in use, Adaptive Server reports Error 701.

Procedure cache sizing

On a production server, minimize the number of procedure reads from disk. When a user needs executes a procedure, Adaptive Server should be able to find an unused tree or plan in the procedure cache for the most common procedures. The percentage of times the server finds an available plan in cache is called the **cache hit ratio**. Keeping a high cache hit ratio for procedures in cache improves performance.

The formulas in Figure 5-2 suggest a good starting point.

Figure 5-2: Formulas for sizing the procedure cache

$$\text{Procedure cache size} = (\text{Max \# of concurrent users}) * (4 + \text{Size of largest plan}) * 1.25$$

$$\text{Minimum procedure cache size needed} = (\text{\# of main procedures}) * (\text{Average plan size})$$

If you have nested stored procedures—procedure A calls procedure B, which calls procedure C—all of them must be in the cache at the same time. Add the sizes for nested procedures, and use the largest sum instead of “Size of largest plan” in the formula in Figure 5-2.

The minimum procedure cache size is the smallest amount of memory that allows at least one copy of each frequently used compiled object to reside in cache. However, the procedure cache can also be used as additional memory at execution time for sorting and query optimization as well as for other purposes. Furthermore, the memory required is based on the type of the query.

Use of sp_monitorconfig to configure procedure cache:

1. Configure procedure cache to the minimum size as determined above.
2. Run your normal database load. If you get error 701, increase procedure cache size. Tune the size of the increase to avoid over-allocation. The recommended increase is (128 * (size of procedure cache, in GB)). For procedure cache size less than 1GB, increase in 128MB increments. For procedure cache size greater than 1GB but less than 2GB, increase in 256MB increments, and so on.
3. Run sp_monitorconfig “procedure cache size” when Adaptive Server has reached or passed the peak load.

4. If sp_monitorconfig indicates that Max_Used is considerably less than the current value for procedure cache from sp_configure, then procedure cache is over-allocated. Consider reducing the procedure cache size configuration value so that a smaller procedure cache may be allocated during the next restart.
5. A value other than zero for the Num_Reuse output from sp_monitorconfig also indicates a shortage of procedure cache. If this value increases over a period of time, consider increasing procedure cache size as suggested in step 2 above.

Estimating stored procedure size

sysprocedures stores the normalized query tree for procedures. Including other overheads, this size allows for 2 rows per 2K page. To estimate the size of a single stored procedure, view, or trigger, use:

```
select count(*) as "approximate size in KB"
from sysprocedures
where id = object_id("procedure_name")
```

For example, to find the size of the titleid_proc in pubs2:

```
select count(*)
from sysprocedures
where id = object_id("titleid_proc")

approximate size in KB
----------------------
                     3
```

If the plan is in cache, the monCachedProcedures monitoring table includes its size.

Estimating the procedure cache size for a sort

To find the size of the procedure cache used for a sort (used by create index, update statistics, order by,distinct, sort and merge join), first determine the number of rows per page:

$$\text{Rows per page} = \frac{\text{Page size}}{\text{minimum length of row}}$$

Determine the procedure cache size used for a sort with this formula:

$$\text{Procedure cache size} = (\text{\# of sort buffers}) \times (\text{rows per page}) \times 85 \text{ bytes}$$

Note If you use a 64-bit system, use 100 bytes in this formula.

Estimating the amount of procedure cache used by create index

create index sorts data within the index key. This sort may require one or more in-memory sorts and one or more disk-based sorts. Adaptive Server loads the data create index sorts into the data cache associated with the table upon which it is creating an index. The number of data cache buffers the sort uses is limited by the value of number of sort buffers. If all keys being sorted fit into the value for number of sort buffers, Adaptive Server peforms a single, in-memory sort operation. If the keys being sorted do not fit into the sort buffers, Adaptive Server must write the results of the in-memory sort to disk so Adaptive Server can load the next set of index keys into the sort buffers to be sorted.

In addition to the sort buffers allocated from the data cache, this sort operation also requires about 66 bytes of metadata from the procedure cache. The formula for the amount of procedure cache used, assuming all of the sort buffers are used, is:

$$\text{Number of 2K procedure cache buffers required} = \frac{(\textit{rows\_per\_page}) \times (\textit{number\_of\_sort\_buffers}) \times 66 \text{ bytes}}{2048 \text{ (assuming 2K page size)}}$$

Example 1

In this example,

- The number of sort buffers set to 500
- create index creates a 15 byte field, yielding 131 rows per page

As a result, all 500 2K buffers are used to hold the data being sorted, and the procedure cache uses 2,111 2K buffers:

(131 X 500 X66) / 2048 = 2,111 2K buffers

Example 2

In this example,

- The number of sort buffers set to 5000

- create index creates a 8 byte field, yielding about 246 rows per page

As a result, all 5,000 2K buffers are used to hold the data being sorted, and the procedure cache uses 39, 639 2K buffers:

(246 X 5,000 X 66) / 2048 = 39,639 2K buffers

Example 3

In this example:

- The number of sort buffers set to 1000
- The table is small, and you can load it completely in to a 800 2K sort buffer data cache, leaving 200 data cache sort buffers available for overhead
- create index creates a 50 byte field, yielding about 39 rows per page

As a result, all 5000 2K buffers are used to hold the data being sorted:

(39 X 1,000 X 66) / 2048 = 1,257 2K buffers

But the data cache has 200 2K buffers left over for overhead, so the procedure cache uses 1057 2K buffers.

Reducing query processing latency

The query processing layer in Adaptive Server 15.7 enables multiple client connections to reuse or share dynamic SQL lightweight procedures (LWPs).

Reusing dynamic SQL LWPs across multiple connections

In versions earlier than 15.7, Adaptive Server stored dynamic SQL statements (prepared statements) and their corresponding LWP in the dynamic SQL cache. Each LWP for a dynamic SQL statement was identified based on the connection metadata. Because connections had different LWPs associated with the same SQL statement, they could not reuse or share the same LWP. In addition, all LWPS and query plans created by the connection were lost when the Dynamic SQL cache was released.

In versions 15.7 and later, Adaptive Server uses the statement cache to also store dynamic SQL statements converted to LWPs. Because the statement cache is shared among all connections, dynamic SQL statements can be reused across connections. These statements are not cached:

- select into statements.
- insert-values statements with all literal values and no parameters.

- Queries that do not reference any tables.
- Individual prepared statements that contain multiple SQL statements. For example:

```
    statement.prepare('insert t1 values (1) insert
    t2 values (3)');
```

- Statements that cause instead-of triggers to fire.

To enable using the statement cache to store dynamic SQL statements, set the enable functionality group or streamlined dynamic SQL configuration options to 1. See "Setting Configuration Parameters" in the *System Administration Guide: Volume 1*.

Using the statement cache has several benefits:

- LWPs and their associated plans are not purged from the statement cache when the connection that created the entry exits.
- LWPs can be shared across connections, further improving performance.
- Reusing LWPs also improves performance in execute cursors.
- Dynamic SQL statements can be monitored from the monitoring table monCachedStatement.

Note Reusing dynamic SQL LWPs may have a negative impact on performance because the reused plan is generated with the original set of supplied parameter values.

Statement cache

The statement cache saves SQL text and plans previously generated for ad hoc SQL statements, enabling Adaptive Server to avoid recompiling incoming SQL that matches a previously cached statement. When enabled, the statement cache reserves a portion of the procedure cache. See the *System Administration Guide: Volume 2* for a complete discussion of the statement cache, including its memory usage.

Data cache

Default data cache and other caches are configured as absolute values. The data cache contains pages from recently accessed objects, typically:

- sysobjects, sysindexes, and other system tables for each database
- Active log pages for each database
- The higher levels and parts of the lower levels of frequently used indexes
- Recently accessed data pages

When you install Adaptive Server, it has a single data cache that is used by all Adaptive Server processes and objects for data, index, and log pages. The default size is 8MB.

The following pages describe the way this single data cache is used. Most of the concepts on aging, buffer washing, and caching strategies apply to the user-defined data caches and the default data cache.

"Configuring the data cache to improve performance" on page 101 describes how to improve performance by dividing the data cache into named caches and how to bind particular objects to these named caches.

Page aging in data cache

The Adaptive Server data cache is managed on a most recently used/least recently used (MRU/LRU) basis. As pages in the cache age, they enter a wash area, where any dirty pages (pages that have been modified while in memory) are written to disk. There are some exceptions to this:

- Caches configured with relaxed LRU replacement policy use the wash section as described above, but are not maintained on an MRU/LRU basis.

  Typically, pages in the wash section are clean; that is, the I/O on these pages has been completed. When a task or query obtains a page from the LRU end, it expects the page to be clean. If not, the query must wait for the I/O to complete on the page, which impairs performance.

- A special strategy ages out index pages and **OAM pages** more slowly than data pages. These pages are accessed frequently in certain applications and keeping them in cache can significantly reduce disk reads.

  See Chapter 10, "Checking Database Consistency," in *System Administration Guide: Volume 2* for more information.

- Adaptive Server may choose to use the LRU cache replacement strategy that does not flush other pages out of the cache with pages that are used only once for an entire query.
- The checkpoint process ensures that, if Adaptive Server needs to be restarted, the recovery process can be completed in a reasonable period of time.

  When the checkpoint process estimates that the number of changes to a database will take longer to recover than the configured value of the recovery interval configuration parameter, it traverses the cache, writing dirty pages to disk.
- Recovery uses only the default data cache, making it faster.
- The housekeeper wash task writes dirty pages to disk when idle time is available between user processes.

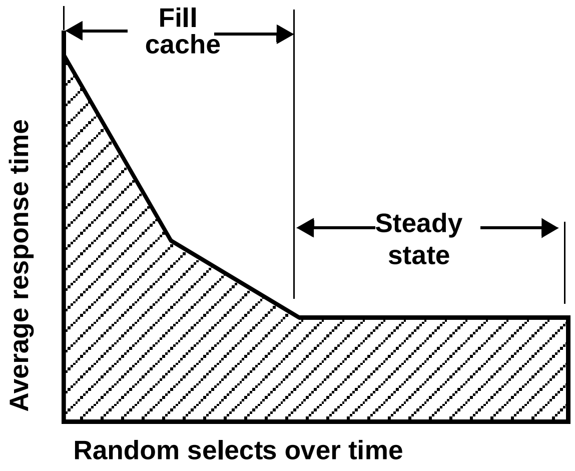
Effect of data cache on retrievals

Figure 5-3 shows the effect of data caching on a series of random select statements executed over a period of time. If the cache is empty initially, the first select statement is guaranteed to require disk I/O. Be sure to adequately size the data cache for the number of transactions you expect against the database.

As more queries are executed and the cache is filled, there is an increasing probability that one or more page requests can be satisfied by the cache, thereby reducing the average response time of the set of retrievals.

When the cache is filled, there is a fixed probability of finding a desired page in the cache from that point forward.

Figure 5-3: Effects of random selects on the data cache



If the cache is smaller than the total number of pages that are being accessed in all databases, there is a chance that a given statement must perform some disk I/O. A cache does not reduce the maximum possible response time—some queries may still need to perform physical I/O for all of the pages they need. But caching decreases the likelihood that the maximum delay will be suffered by a particular query—more queries are likely to find at least some of the required pages in cache.

Effect of data modifications on the cache

The behavior of the cache in the presence of update transactions is more complicated than for retrievals.

There is still an initial period during which the cache fills. Then, because cache pages are being modified, there is a point at which the cache must begin writing those pages to disk before it can load other pages. Over time, the amount of writing and reading stabilizes, and subsequent transactions have a given probability of requiring a disk read and another probability of causing a disk write.

The steady-state period is interrupted by checkpoints, which cause the cache to write all dirty pages to disk.

Data cache performance

You can observe data cache performance by examining the **cache hit ratio**, the percentage of page requests that are serviced by the cache.

One hundred percent is outstanding, but implies that your data cache is as large as the data or at least large enough to contain all the pages of your frequently used tables and indexes.

A low percentage of cache hits indicates that the cache may be too small for the current application load. Very large tables with random page access generally show a low cache hit ratio.

Testing data cache performance

Consider the behavior of the data and procedure caches when you measure the performance of a system. When a test begins, the cache can be in any one of the following states:

- Empty
- Fully randomized
- Partially randomized
- Deterministic

An empty or fully randomized cache yields repeatable test results because the cache is in the same state from one test run to another.

A partially randomized or deterministic cache contains pages left by transactions that were just executed. Such pages could be the result of a previous test run. In these cases, if the next test steps request those pages, then no disk I/O is needed.

Such a situation can bias the results away from a purely random test and lead to inaccurate performance estimates.

The best testing strategy is to start with an empty cache or to make sure that all test steps access random parts of the database. For precise testing, execute a mix of queries that is consistent with the planned mix of user queries on your system.

Cache hit ratio for a single query

To see the cache hit ratio for a single query, use set statistics io on to see the number of logical and physical reads, and set showplan on to see the I/O size used by the query.

Figure 5-4: Formula to compute cache hit ratio

$$\text{Cache hit ratio} = \frac{\text{Logical reads} - (\text{Physical reads} * \text{Pages})}{\text{Logical reads}}$$

With statistics io, physical reads are reported in I/O-size units. If a query uses 16K I/O, it reads 8 pages with each I/O operation.

If statistics io reports 50 physical reads, it has read 400 pages. Use showplan to see the I/O size used by a query.

Cache hit ratio information from *sp_sysmon*

sp_sysmon reports on cache hits and misses for:

- All caches on Adaptive Server
- The default data cache
- Any user-configured caches

The server-wide report provides the total number of cache searches and the percentage of cache hits and cache misses.

For each cache, the report contains the number of cache searches, cache hits, and cache misses, and the number of times that a needed buffer was found in the wash section.

See *Performance and Tuning Series: Monitoring Adaptive Server with sp_sysmon*.

Configuring the data cache to improve performance

When you install Adaptive Server, it has single default data cache, with a 2K memory pool, one cache partition, and a single spinlock.

To improve performance, add data caches and bind databases or database objects to them:

1. To reduce contention on the default data cache spinlock, divide the cache into *n* where *n* is 1, 2, 4, 8,16, 32 or 64. If you have contention on the spinlock (designated here with “x”) with one cache partition, the spinlock contention is expected to reduce to x/*n*, where *n* is the number of partitions.
2. When a particular cache partition spinlock table that is hot—a table in high demand by user applications—consider splitting the default cache into named caches.
3. If there is still contention, consider splitting the named cache into named cache partitions.

You can configure 4K, 8K, and 16K buffer pools from the logical page size in both user-defined data caches and the default data caches, allowing Adaptive Server to perform large I/O. In addition, caches that are sized to completely hold tables or indexes can use the relaxed LRU cache policy to reduce overhead.

You can also split the default data cache or a named cache into partitions to reduce spinlock contention.

Try configuring the data cache for improved performance in these ways:

- Configure named data caches to be large enough to hold critical tables and indexes. This keeps other server activity from contending for cache space and speeds queries using these tables, since the needed pages are always found in cache.

  You can configure these caches to use the relaxed LRU replacement policy, reducing the cache overhead.
- To increase concurrency, bind a hot table to one cache and the indexes on the table to other caches.
- Create a named data cache large enough to hold the hot pages of a table where a high percentage of the queries reference only a portion of the table.

  For example, if a table contains data for a year, but 75% of the queries reference data from the most recent month (about 8% of the table), configuring a cache of about 10% of the table size provides room to keep the most frequently used pages in cache and leaves some space for the less frequently used pages.
- Assign tables or databases used in decision-support systems (DSS) to specific caches with large I/O configured.

This keeps DSS applications from contending for cache space with OLTP applications. DSS applications typically access large numbers of sequential pages, and OLTP applications typically access relatively few random pages.

- Bind tempdb to its own cache to keep it from contending with other user processes.

  Proper sizing of the tempdb cache can keep most tempdb activity in memory for many applications. If this cache is large enough, tempdb activity can avoid performing I/O.

- Bind text pages to named caches to improve the performance on text access.
- Bind a database's log to a cache, again reducing contention for cache space and access to the cache.
- When a user process makes changes to a cache, a spinlock denies all other processes access to the cache.

  Although spinlocks are held for extremely brief durations, they can slow performance in multiprocessor systems with high transaction rates. When you configure multiple caches, each cache is controlled by a separate spinlock, increasing concurrency on systems with multiple CPUs.

  Within a single cache, adding cache partitions creates multiple spinlocks to further reduce contention. Spinlock contention is not an issue on single-engine servers.

Most of these possible uses for named data caches have the greatest impact on multiprocessor systems with high transaction rates or with frequent DSS queries and multiple users. Some of them can increase performance on single CPU systems when they lead to improved utilization of memory and reduce I/O.

Commands to configure named data caches

The commands used to configure caches and pools are shown in Table 5-1

Table 5-1: Commands used to configure caches

Command	Function
sp_cacheconfig	Creates or drops named caches and set the size, cache type, cache policy and local cache partition number. Reports on sizes of caches and pools.

Command	Function
sp_poolconfig	Creates and drops I/O pools and changes their size, wash size, and asynchronous prefetch limit.
sp_bindcache	Binds databases or database objects to a cache.
sp_unbindcache	Unbinds the specified database or database object from a cache.
sp_unbindcache_all	Unbinds all databases and objects bound to a specified cache.
sp_helpcache	Reports summary information about data caches and lists the databases and database objects that are bound to a cache. Also reports on the amount of overhead required by a cache.
sp_sysmon	Reports statistics useful for tuning cache configuration, including cache spinlock contention, cache utilization, and disk I/O patterns.

For a full description of configuring named caches and binding objects to caches, see Chapter 4, "Configuring Data Caches," in *System Administration Guide: Volume2*. Only a system administrator can configure named caches and bind database objects to them.

Tuning named caches

Creating named data caches and memory pools, and binding databases and database objects to the caches, can dramatically hurt or improve Adaptive Server performance. For example:

- A cache that is poorly used hurts performance.

  If you allocate 25% of your data cache to a database that services a very small percentage of the query activity on your server, I/O increases in other caches.

- An unused pool hurts performance.

  If you add a 16K pool, but none of your queries use it, you have taken space away from the 2K pool. The 2K pool's cache hit ratio is reduced, and I/O is increased.

- An overused pool hurts performance.

  If you configure a 16K pool, and virtually all of your queries use it, I/O rates are increased. The 2K cache will be under used, while pages are rapidly cycled through the 16K pool. The cache hit ratio in the 16K pool will be very poor.

- When you balance pool usage within a cache, performance can increase dramatically.

Both 16K and 2K queries experience improved cache hit ratios. The large number of pages often used by queries that perform 16K I/O do not flush 2K pages from disk. Queries using 16K perform approximately one-eighth the number of I/Os required by 2K I/O.

When tuning named caches, always measure current performance, make your configuration changes, and measure the effects of the changes with similar workload.

Cache configuration goals

Goals for configuring caches include:

- Reduced contention for spinlocks on multiple engine servers.
- Improved cache hit ratios and reduced disk I/O. As a bonus, improving cache hit ratios for queries can reduce lock contention, since queries that do not need to perform physical I/O usually hold locks for shorter periods of time.
- Fewer physical reads, due to the effective use of large I/O.
- Fewer physical writes, because recently modified pages are not flushed from cache by other processes.
- Reduced cache overhead and reduced CPU bus latency on SMP systems, when relaxed LRU policy is appropriately used.
- Reduced cache spinlock contention on SMP systems, when cache partitions are used.

In addition to commands such as showplan and statistics io that help tune on a per-query basis, use a performance monitoring tool such as sp_sysmon to look at how multiple queries and multiple applications share cache space when they run simultaneously.

Gather data, plan, and then implement

The first step in developing a plan for cache usage is to provide as much memory as possible for the data cache:

- Determine the maximum amount of memory you can allocate to Adaptive Server. Set max memory to that value.

- After you have set all the parameters that use Adaptive Server memory, the difference between max memory and the run value of total logical memory is the memory available for additional configuration and for data and procedure caches. If you have sufficiently configured all the other configuration parameters, you can allocate this additional memory to data caches. Most changes to the data cache are dynamic and do not require a restart.
- If you allocate all the additional memory to data caches, there may not be any memory available to reconfigure other configuration parameters. However, if there is additional memory available, you can dynamically increase max memory and other dynamic configuration parameters like procedure cache size, user connections, and so on.
- Use your performance monitoring tools to establish baseline performance, and to establish your tuning goals.

Determine the size of memory you can allocate to data caches, as mentioned in the above steps. Include the size of already configured caches, like the default data cache and any named caches.

Determine data cache size by looking at existing objects and applications. Adding new caches or increasing configuration parameters that consume memory does not reduce the size of the default data cache. When you have decided what memory is available for data caches and the size of each individual cache, add new caches and increase or decrease size of existing data caches.

- Evaluate cache needs by analyzing I/O patterns, and evaluate pool needs by analyzing query plans and I/O statistics.
- Configure the easiest choices that will gain the most performance first:
    - Choose a size for a tempdb cache.
    - Choose a size for any log caches, and tune the log I/O size.
    - Choose a size for the specific tables or indexes that you want to keep entirely in cache.
    - Add large I/O pools for index or data caches, as appropriate.
- After you determine these sizes, examine remaining I/O patterns, cache contention, and query performance. Configure caches proportional to I/O usage for objects and databases.

Keep performance goals in mind as you configure caches:

- If the major goal is to reduce spinlock contention, increasing the number of cache partitions for heavily used caches may be the only step.

  Moving a few high-I/O objects to separate caches also reduces spinlock contention and improves performance.
- If the major goal is to improve response time by improving cache hit ratios for particular queries or applications, creating caches for the tables and indexes used by those queries should be guided by a thorough understanding of access methods and I/O requirements.

Evaluating cache needs

Generally, configure caches in proportion to the number of times that the pages in the caches will be accessed by queries, and configure pools within caches in proportion to the number of pages used by queries that choose I/O of that pool's size.

If your databases and their logs are on separate logical devices, you can estimate cache proportions using sp_sysmon or operating system commands to examine physical I/O by device.

See *Performance and Tuning Series: Monitoring Adaptive Server with sp_sysmon*.

Large I/O and performance

You can configure the default cache and any named caches you create for large I/O by splitting a cache into pools. The default I/O size is 2K, one Adaptive Server data page.

Note Reference to large I/Os are on a 2K logical page size server. If you have an 8K page size server, the basic unit for the I/O is 8K. If you have a 16K page size server, the basic unit for the I/O is 16K.

For queries where pages are stored and accessed sequentially, Adaptive Server reads up to eight data pages in a single I/O. Since the majority of I/O time is spent doing physical positioning and seeking on the disk, large I/O can greatly reduce disk access time. In most cases, configure a 16K pool in the default data cache.

Certain types of Adaptive Server queries are likely to benefit from large I/O. Identifying these queries can help determine the correct size for data caches and memory pools.

In the following examples, either the database or the specific table, index, or large object (LOB) page change (used for text, image, and Java off-row columns) must be bound to a named data cache that has large memory pools, or the default data cache must have large I/O pools. Types of queries that can benefit from large I/O include:

- Queries that scan entire tables. For example:

  ```
  select title_id, price from titles
  select count(*) from authors
      where state = "CA"   /* no index on state */
  ```

- Range queries on tables with clustered indexes. For example:

  ```
  where indexed_colname >= value
  ```

- Queries that scan the leaf level of an index, both matched and unmatched scans. If there is a nonclustered index on type, price, this query could use large I/O on the leaf level of the index, since all the columns used in the query are contained in the index:

  ```
  select type, sum(price)
      from titles
      group by type
  ```

- Queries that join entire tables, or large portions of tables. Different I/O sizes may be used on different tables in a join.
- Queries that select text or image or Java off-row columns. For example:

  ```
  select au_id, copy from blurbs
  ```

- Queries that generate Cartesian products. For example:

  ```
  select title, au_lname
       from titles, authors
  ```

  This query needs to scan all of one table, and scan the other table completely for each row from the first table. Caching strategies for these queries follow the same principles as for joins.

- Queries such as select into that allocate large numbers of pages.

Note Adaptive Server version 12.5.0.3 or later enables large-page allocation in select into. It allocates pages by extent rather than by individual page, thus issuing fewer logging requests for the target table.

If you configure Adaptive Server with large buffer pools, it uses large I/O buffer pools when writing the target table pages to disk.

- create index commands.
- Bulk copy operations on heaps—both copy in and copy out.
- The update statistics, dbcc checktable, and dbcc checkdb commands.

The optimizer and cache choices

If the cache for a table or index has a 16K pool, the optimizer determines the I/O size to use for data and leaf-level index pages based on the number of pages that must be read, and the cluster ratios for the table or index.

The optimizer's knowledge is limited to the single query it is analyzing and to statistics about the table and cache. It does not know how many other queries are simultaneously using the same data cache. It has no statistics on whether table storage is fragmented such that large I/Os or asynchronous prefetch would be less effective.

In some cases, this combination of factors can lead to excessive I/O. For example, users may experience higher I/O and poor performance if simultaneous queries with large result sets are using a very small memory pool.

Choosing the right mix of I/O sizes for a cache

You can configure up to four pools in any data cache, but, in most cases, caches for individual objects perform best with only a 2K pool and a 16K pool. A cache for a database where the log is not bound to a separate cache should also have a pool configured to match the log I/O size configured for the database; often the best log I/O size is 4K.

Reducing spinlock contention with cache partitions

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.

The number of cache partitions is always a power of 2. Each time you increase the number of partitions, you reduce the spinlock contention by approximately 1/2. If spinlock contention is greater than 10 to 15%, consider increasing the number of partitions for the cache. This example creates 4 partitions in the default data cache:

```
sp_cacheconfig "default data cache",
"cache_partition=4"
```

You must restart the server for changes in cache partitioning to take effect.

See Chapter 4, “Configuring Data Caches,” in *System Administration Guide: Volume 2*.

For information on monitoring cache spinlock contention with sp_sysmon, see *Performance and Tuning Series: Monitoring Adaptive Server with sp_sysmon*.

Each pool in the cache is partitioned into a separate LRU/MRU chain of pages, with its own wash marker.

Cache replacement strategies and policies

The Adaptive Server optimizer uses two cache replacement strategies to keep frequently used pages in cache while flushing the less frequently used pages. To reduce cache overhead, you may want to consider setting the cache replacement policy for some caches.

Strategies

Replacement strategies determine where the page is placed in cache when it is read from disk. The optimizer decides on the cache replacement strategy to be used for each query:

- The fetch-and-discard, or MRU replacement, strategy links the newly read buffers at the wash marker in the pool.
- The LRU replacement strategy links newly read buffers at the most-recently used (MRU) end of the pool.

Cache replacement strategies can affect the cache hit ratio for your query mix:

- Pages that are read into cache with the fetch-and-discard strategy remain in cache a much shorter time than queries read in at the MRU end of the cache. If such a page is needed again (for example, if the same query is run again very soon), the pages will probably need to be read from disk again.
- Pages that are read into cache with the fetch-and-discard strategy do not displace pages that already reside in cache before the wash marker. This means that the pages already in cache before the wash marker are not flushed out of cache by pages that are needed only once by a query.

See Chapter 7, “Controlling Optimization,” in *Performance and Tuning Series: Query Processing and Abstract Plans*.

Policies

A system administrator can specify whether a cache is going to be maintained as an MRU/LRU-linked list of pages (strict) or whether relaxed LRU replacement policy can be used.

- **Strict replacement policy** – replaces the least recently used page in the pool, linking the newly read pages at the beginning (MRU end) of the page chain in the pool.
- **Relaxed replacement policy** – attempts to avoid replacing a recently used page, but without the overhead of keeping buffers in LRU/MRU order.

The default cache replacement policy is strict replacement. 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.

Relaxed LRU policy saves the overhead of maintaining the cache in MRU/LRU order. On SMP systems, where copies of cached pages may reside in hardware caches on the CPUs themselves, relaxed LRU policy can reduce bandwidth on the bus that connects the CPUs.

If you have created a cache to hold all of, or most, certain objects, and the cache hit rate is above 95%, using relaxed cache replacement policy for the cache can improve performance slightly.

See Chapter 4, "Configuring Data Caches," in *System Administration Guide: Volume2*.

Configuring relaxed LRU replacement for database logs

Log pages are filled with log records and are immediately written to disk. When applications include triggers, deferred updates, or transaction rollbacks, some log pages may be read, but usually they are very recently used pages, which are still in the cache.

Since accessing these pages in cache moves them to the MRU end of a strict-replacement policy cache, log caches may perform better with relaxed LRU replacement.

Relaxed LRU replacement for lookup tables and indexes

User-defined caches that are sized to hold indexes and frequently used lookup tables are good candidates for relaxed LRU replacement. If a cache is a good candidate, but you find that the cache hit ratio is slightly lower than the performance guideline of 95%, determine whether slightly increasing the size of the cache can provide enough space to completely hold the table or index.

Named data cache recommendations

These cache recommendations can improve performance on both single and multiprocessor servers:

- Because Adaptive Server writes log pages according to the size of the logical page size, larger log pages potentially reduce the rate of commit-sharing writes for log pages.

Commit-sharing occurs when, instead of performing many individual commits, Adaptive Server waits until it can perform a batch of commits. Per-process user log caches are sized according to the logical page size and the user log cache size configuration parameter. The default size of the user log cache is one logical page.

For transactions that generate many log records, the time required to flush the user log cache is slightly higher for larger logical page sizes. However, because the log-cache sizes are also larger, Adaptive Server does not need to perform as many log-cache flushes to the log page for long transactions.

See Chapter 4, "Configuring Data Caches," in *System Administration Guide: Volume2*.

- Create a named cache for tempdb and configure the cache for 16K I/O for use by select into queries and sorts.
- Create a named cache for the logs for high-use databases. Configure pools in this cache to match the log I/O size set with sp_logiosize.

  See "Choosing the I/O size for the transaction log" on page 116.
- If a table or its index is small and constantly in use, create a cache for only that object or for a few objects.
- For caches with cache hit ratios of more than 95%, configure relaxed LRU cache replacement policy if you are using multiple engines.
- Keep cache sizes and pool sizes proportional to the cache utilization objects and queries:
  - If 75% of the work on your server is performed in one database, that allocate approximately 75% of the data cache, in a cache created specifically for the database, in caches created for its busiest tables and indexes, or in the default data cache.
  - If approximately 50% of the work in your database can use large I/O, configure about 50% of the cache in a 16K memory pool.
- View the cache as a shared resource rather than attempt to micromanage the caching needs of every table and index.

  Start cache analysis and testing at the database level, concentrating on particular tables and objects with high I/O needs or high application priorities and those with special uses, such as tempdb and transaction logs.
- On SMP servers, use multiple caches to avoid contention for the cache spinlock:

- Use a separate cache for the transaction log for busy databases, and use separate caches for some of the tables and indexes that are accessed frequently.
- If spinlock contention is greater than 10% on a cache, split it into multiple caches or use cache partitions.

  Use sp_sysmon periodically during high-usage periods to check for cache contention. See *Performance and Tuning Series: Monitoring Adaptive Server with sp_sysmon.*
- Set relaxed LRU cache policy on caches with cache hit ratios of more than 95%, such as those configured to hold an entire table or index.

Sizing caches for special objects, *tempdb*, and transaction logs

Creating caches for tempdb, the transaction logs, and for a few tables or indexes that you want to keep completely in cache can reduce cache spinlock contention and improve cache hit ratios.

Use sp_spaceused to determine the size of the tables or indexes that you want to keep entirely in cache. If you know how fast these tables increase in size, allow some extra cache space for their growth. To see the size of all the indexes for a table, use:

```
sp_spaceused table_name, 1
```

Examining cache needs for *tempdb*

Look at the use of tempdb:

- Estimate the size of the temporary tables and worktables generated by your queries.

  Look at the number of pages generated by select into queries.

  These queries can use 16K I/O, so you can use this information to help you size a 16K pool for the tempdb cache.
- Estimate the duration (in wall-clock time) of the temporary tables and worktables.
- Estimate how often queries that create temporary tables and worktables are executed.

- Try to estimate the number of simultaneous users, especially for queries that generate very large result sets in tempdb.

With this information, you can a form a rough estimate of the amount of simultaneous I/O activity in tempdb. Depending on your other cache needs, you can size tempdb so that virtually all tempdb activity takes place in cache, and few temporary tables are actually written to disk.

In most cases, the first 2MB of tempdb are stored on the master device, with additional space allocated to a logical device. Use sp_sysmon to check those devices to help determine physical I/O rates.

Examining cache needs for transaction logs

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.

The current page of the transaction log is written to disk when transactions commit, so try to size the log to reduce the number of times that processes that need to reread log pages must go to disk because the pages have been flushed from the cache.

These Adaptive Server processes need to read log pages:

- Triggers that use the inserted and deleted tables, which are built from the transaction log when the trigger queries the tables
- Deferred updates, deletes, and inserts, since these require rereading the log to apply changes to tables or indexes
- Transactions that are rolled back, since log pages must be accessed to roll back the changes

When sizing a cache for a transaction log:

- Examine the duration of processes that need to reread log pages.

  Estimate how long the longest triggers and deferred updates last.

  If some of your long-running transactions are rolled back, check the length of time they ran.
- Check transaction log size with sp_spaceused at regular intervals to estimate how fast the log grows.

Use this log growth estimate and the time estimate to size the log cache. For example, if the longest deferred update takes 5 minutes, and the transaction log for the database grows at 125 pages per minute, 625 pages are allocated for the log while this transaction executes.

If a few transactions or queries are especially long-running, you may want to size the log for the average, rather than the maximum, length of time.

Choosing the I/O size for the transaction log

When a user performs operations that require logging, log records are first stored in a user log cache until events flush the user's log records to the current transaction log page in cache. Log records are flushed when:

- A transaction ends
- The user log cache is full
- The transaction changes tables in another database
- Another process needs to write a page referenced in the user log cache
- Certain system events occur

To economize on disk writes, Adaptive Server holds partially filled transaction log pages for a very brief span of time so that records of several transactions can be written to disk simultaneously. This process is called group commit.

In environments with high transaction rates or with transactions that create large log records, the 2K transaction log pages fill quickly. A large proportion of log writes are due to full log pages, rather than group commits.

Creating a 4K pool for the transaction log can greatly reduce the number of log writes in these environments.

sp_sysmon reports on the ratio of transaction log writes to transaction log allocations. Try using 4K log I/O if all these conditions are true:

- The database uses 2K log I/O.
- The number of log writes per second is high.
- The average number of writes per log page is slightly more than one.

Here is some sample output showing that a larger log I/O size might help performance:

```
                         per sec   per xact    count  % of total
Transaction Log Writes      22.5       458.0      1374     n/a
Transaction Log Alloc       20.8       423.0      1269     n/a
```

```
Avg # Writes per Log Page    n/a          n/a   1.08274     n/a
```

See *Performance and Tuning Series: Monitoring Adaptive Server with sp_sysmon.*

Configuring for large log I/O size

The log I/O size for each database is reported in the server's error log when Adaptive Server starts. You can also use sp_logiosize.

To see the size for the current database, execute sp_logiosize with no parameters. To see the size for all databases on the server and the cache in use by the log, use:

```
sp_logiosize "all"
```

To set the log I/O size for a database to 4K, the default, you must be using the database. This command sets the size to 4K:

```
sp_logiosize "default"
```

If no 4K pool is available in the cache used by the log, 2K I/O is used instead.

If a database is bound to a cache, all objects not explicitly bound to other caches use the database's cache. This includes the syslogs table.

To bind syslogs to another cache, you must first put the database in single-user mode, with sp_dboption, and then use the database and execute sp_bindcache:

```
sp_bindcache pubs_log, pubtune, syslogs
```

Additional tuning tips for log caches

For further tuning after configuring a cache for the log, check sp_sysmon output for:

- The cache used by the log
- The disk the log is stored on
- The average number of writes per log page

When looking at the log cache section, check "Cache Hits" and "Cache Misses" to determine whether most of the pages needed for deferred operations, triggers, and rollbacks are being found in cache.

In the "Disk Activity Detail" section, look at the number of "Reads" performed to see how many times tasks that need to reread the log had to access the disk.

Basing data pool sizes on query plans and I/O

Divide a cache into pools based on the proportion of the I/O performed by queries that use the corresponding I/O sizes. If most queries can benefit from 16K I/O, and you configure a very small 16K cache, you may see worse performance.

Most of the space in the 2K pool remains unused, and the 16K pool experiences high turnover. The cache hit ratio is significantly reduced.

The problem is most severe with nested-loop join queries that must repeatedly reread the inner table from disk.

Making a good choice about pool sizes requires:

- Knowledge of the application mix and the I/O size your queries can use
- Careful study and tuning, using monitoring tools to check cache utilization, cache hit rates, and disk I/O

Checking I/O size for queries

You can examine query plans and I/O statistics to determine which queries are likely to perform large I/O, and the amount of I/O those queries perform. This information can form the basis for estimating the amount of 16K I/O the queries should perform with a 16K memory pool. I/Os are done in terms of logical page sizes; if large I/O uses 2K pages, it retrieves in 2K sizes, if 8K pages, it retrieves in the 8K size, as shown:

Logical page size	Memory pool
2K	16K
4K	32K
8K	64K
16K	128K

For another example, consider that a query that scans a table and performs 800 physical I/Os using a 2K pool should perform about 100 8K I/Os.

See “Large I/O and performance” on page 107 for a list of query types.

To test your estimates, configure the pools and run individual queries and your target mix of queries to determine optimum pool sizes. Choosing a good initial size for your first test using 16K I/O depends on a good sense of the types of queries in your application mix.

This estimate is especially important if you are configuring a 16K pool for the first time on an active production server. Make the best possible estimate of simultaneous uses of the cache.

These guidelines provide some points of reference:

- If most I/O occurs in point queries using indexes to access a small number of rows, make the 16K pool relatively small, perhaps 10 to 20% of the cache size.
- If you estimate that a large percentage of the I/Os will use the 16K pool, configure 50 to 75% of the cache for 16K I/O.

  Queries that use 16K I/O include any query that scans a table, uses the clustered index for range searches and order by, and queries that perform matching or nonmatching scans on covering nonclustered indexes.
- If you are unsure about the I/O size that will be used by your queries, configure about 20% of your cache space in a 16K pool, and use showplan and statistics i/o while you run your queries.

  Examine the showplan output for the "Using 16K I/O" message. Check statistics i/o output to see how much I/O is performed.
- If you think that your typical application mix uses both 16K I/O and 2K I/O simultaneously, configure 30 to 40% of your cache space for 16K I/O.

  Your optimum may be higher or lower, depending on the actual mix and the I/O sizes chosen by the query.

  If many tables are accessed by both 2K I/O and 16K I/O, Adaptive Server cannot use 16K I/O, if any page from the extent is in the 2K cache. It performs 2K I/O on the other pages in the extent that are needed by the query. This adds to the I/O in the 2K cache.

  After configuring for 16K I/O, check cache usage and monitor the I/O for the affected devices, using sp_sysmon or Adaptive Server Monitor. Also, use showplan and statistics io to observe your queries.

  - Look for nested-loop join queries where an inner table would use 16K I/O, and the table is repeatedly scanned using the fetch-and-discard (MRU) strategy.

    This can occur when neither the outer or inner table fits completely in cache. You can significantly reduce I/O by increasing the size of the 16K pool to allow the inner table to fit completely in cache. You might also consider binding the two tables to separate caches.
- Look for excessive 16K I/O, when compared to table size in pages.

For example, if you have an 8000-page table, and a 16K I/O table scan performs significantly more than 1000 I/Os to read this table, you may see improvement by re-creating the clustered index on this table.

- Look for times when large I/O is denied. Many times, this is because pages are already in the 2K pool, so the 2K pool is used for the rest of the I/O for the query.

  See Chapter 7, “Controlling Optimization,” in *Performance and Tuning Series: Query Processing and Abstract Plans*.

Configuring buffer wash size

You can configure the wash area for each pool in each cache. If you set the wash size is set too high, Adaptive Server may perform unnecessary writes. If you set the wash area too small, Adaptive Server may not be able to find a clean buffer at the end of the buffer chain and may have to wait for I/O to complete before it can proceed. Generally, wash size defaults are correct and need to be adjusted only in large pools that have very high rates of data modification.

Adaptive Server allocates buffer pools in units of logical pages. For example, on a server using 2K logical pages, 8MB is allocated to the default data cache. By default, this constitutes approximately 4096 buffers.

If you allocate the same 8MB for the default data cache on a server using a 16K logical page size, the default data cache is approximately 512 buffers. On a busy system, this small number of buffers might result in a buffer always being in the wash region, causing a slowdown for tasks requesting clean buffers.

In general, to obtain the same buffer management characteristics on larger page sizes as with 2K logical page sizes, scale the cache size to the larger page size. In other words, if you increase logical page size by four times, increase cache and pool sizes by about four times larger as well.

Queries performing large I/O, extent-based reads and writes, and so on, benefit from the use of larger logical page sizes. However, as catalogs continue to be page-locked, there is greater contention and blocking at the page level on system catalogs.

Row and column copying for data-only locked tables results in a greater slowdown when used for wide columns. Memory allocation to support wide rows and wide columns marginally slows the server.

See *Performance and Tuning Series: Locking and Concurrency Control*.

Overhead of pool configuration and binding objects

Configuring memory pools and binding objects to caches can affect users on a production system, so perform these activities during off-hours if possible.

Pool configuration overhead

When a pool is created, deleted, or changed, the plans of all stored procedures and triggers that use objects bound to the cache are recompiled the next time they are run. If a database is bound to the cache, this affects all of the objects in a database.

There is a slight amount of overhead involved in moving buffers between pools.

Cache binding overhead

When you bind or unbind an object, all the object's pages that are currently in the cache are flushed to disk (if dirty) or dropped from the cache (if clean) during the binding process.

The next time the pages are needed by user queries, they must be read from the disk again, slowing the performance of the queries.

Adaptive Server acquires an exclusive lock on the table or index while the cache is being cleared, so binding can slow access to the object by other users. The binding process may have to wait to acquire the lock until transactions complete.

Note Binding and unbinding objects from caches removes them from memory. This can be useful when you are tuning queries during development and testing.

If you need to check physical I/O for a particular table, and earlier tuning efforts have brought pages into cache, you can unbind and rebind the object. The next time the table is accessed, all pages used by the query must be read into the cache.

The plans of all stored procedures and triggers using the bound objects are recompiled the next time they are run. If a database is bound to the cache, this affects all the objects in the database.

Maintaining data cache performance for large I/O

When heap tables, clustered indexes, or nonclustered indexes are newly created, they show optimal performance when large I/O is being used. Over time, the effects of deletes, page splits, and page deallocation and reallocation can increase the cost of I/O. optdiag reports a statistic called "Large I/O efficiency" for tables and indexes.

A large I/O is very efficient when this value is 1, or close to 1. As the value decreases, more I/O is required to access data pages needed for a query, and large I/O may be bringing pages into cache that are not needed by the query.

Consider rebuilding indexes when large I/O efficiency drops or when activity in the pool increases due to increased 16K I/O.

When large I/O efficiency decreases, you can:

- Run reorg rebuild on tables that use data-only-locking. You can also use reorg rebuild on the index of data-only-locked tables.
- For allpages-locked tables, drop and re-create the indexes.

See Chapter 6, "Database Maintenance," in *Performance and Tuning Series: Physical Database Tuning*.

Diagnosing excessive I/O counts

There are several reasons why a query that performs large I/O might require more reads than you anticipate:

- The cache used by the query has a 2K cache and other processes have brought pages from the table into the 2K cache.

  If Adaptive Server finds one of the pages it would read using 16K I/Os already in the 2K cache, it performs 2K I/O on the other pages in the extent that are required by the query.

- The first extent on each allocation unit stores the allocation page, so if a query needs to access all the pages on the extent, it must perform 2K I/O on the 7 pages that share the extent with the allocation page.

  The other 31 extents can be read using 16K I/O. The minimum number of reads for an entire allocation unit is always 38, not 32.

- In nonclustered indexes and clustered indexes on data-only-locked tables, an extent may store both leaf-level pages and pages from higher levels of the index. 2K I/O is performed on the higher levels of indexes, and on leaf-level pages when few pages are needed by a query.

  When a covering leaf-level scan performs 16K I/O, it is likely that some of the pages from some extents will be in the 2K cache. The rest of the pages in the extent will be read using 2K I/O.

Using *sp_sysmon* to check large I/O performance

The sp_sysmon output for each data cache includes information that can help you determine the effectiveness for large I/Os. See *Performance and Tuning Series: Monitoring Adaptive Server with sp_sysmon*, and Chapter 7, “Controlling Optimization,” in *Performance and Tuning Series: Query Processing and Abstract Plans*.

- “Large I/O usage” reports the number of large I/Os performed and denied and provides summary statistics.
- “Large I/O detail” reports the total number of pages that were read into the cache by a large I/O and the number of pages that were actually accessed while they were in the cache.

Speed of recovery

As users modify data in Adaptive Server, only the transaction log is written to disk immediately, to ensure that given data or transactions can be recovered. The changed or “dirty” data and index pages stay in the data cache until one of these events causes them to be written to disk:

- The checkpoint process wakes up, determines that the changed data and index pages for a particular database need to be written to disk, and writes out all the dirty pages in each cache used by the database.

  The combination of the setting for recovery interval and the rate of data modifications on your server determine how often the checkpoint process writes changed pages to disk.
- As pages move into the buffer wash area of the cache, dirty pages are automatically written to disk.

- Adaptive Server has spare CPU cycles and disk I/O capacity between user transactions, and the housekeeper wash task uses this time to write dirty buffers to disk.
- Recovery happens only on the default data cache.
- A user issues a checkpoint command.

  You can use checkpoint to identify one or more databases or use an all clause.

  ```
  checkpoint [all | [dbname[, dbname[, dbname.....]]]
  ```

The combination of checkpoints, the housekeeper, and writes started at the wash marker has these benefits:

- Many transactions may change a page in the cache or read the page in the cache, but only one physical write is performed.
- Adaptive Server performs many physical writes when the I/O does not cause contention with user processes.

Tuning the recovery interval

The default recovery interval in Adaptive Server is five minutes per database. Changing the recovery interval affects performance because it impacts the number of times Adaptive Server writes pages to disk.

Table 5-2 shows the effects of changing the recovery interval from its current setting on your system.

Table 5-2: Effects of recovery interval on performance and recovery time

Setting	Effects on performance	Effects on recovery
Lower	May cause more reads and writes and may lower throughput. Adaptive Server writes dirty pages to the disk more often. Any checkpoint I/O spikes will be smaller.	Setting the recovery interval lower expedites recovery if there are no long-running open transactions that Adaptive Server must roll back. If there are long-running open transactions, more frequent checkpoints could slow the recovery process because the disks contains more modifications that Adaptive Server must roll back.

Setting	Effects on performance	Effects on recovery
Higher	Minimizes writes and improves system throughput. Checkpoint I/O spikes will be higher.	Automatic recovery may take more time on start-up. Adaptive Server may have to reapply a large number of transaction log records to the data pages.

See Chapter 11, "Developing a Backup and Recovery Plan," in *System Administration Guide: Volume 2* for information on setting the recovery interval. sp_sysmon reports the number and duration of checkpoints. See *Performance and Tuning Series: Monitoring Adaptive Server with sp_sysmon.*

Effects of the housekeeper wash task on recovery time

Adaptive Server's housekeeper wash task automatically begins cleaning dirty buffers during the server's idle cycles. If the task can flush all active buffer pools in all configured caches, it wakes up the checkpoint process. This may result in faster checkpoints and shorter database recovery time.

System administrators can use the housekeeper free write percent configuration parameter to tune or disable the housekeeper wash task. This parameter specifies the maximum percentage by which the housekeeper task can increase database writes.

For more information on tuning the housekeeper and the recovery interval, see *Performance and Tuning Series: Monitoring Adaptive Server with sp_sysmon*.

Auditing and performance

Heavy auditing can affect performance as follows:

- Audit records are written first to a queue in memory and then to the sybsecurity database. If the database shares a disk used by other busy databases, it can slow performance.
- If the in-memory audit queue fills up, the user processes that generate audit records sleep. See Figure 5-5 on page 127.

Sizing the audit queue

The size of the audit queue can be set by a system security officer. The default configuration is as follows:

- A single audit record requires a minimum of 32 bytes, up to a maximum of 424 bytes.

  This means that a single data page stores between 4 and 80 records.

- The default size of the audit queue is 100 records, requiring approximately 42K.

  The minimum size of the queue is 1 record; the maximum size is 65,335 records.

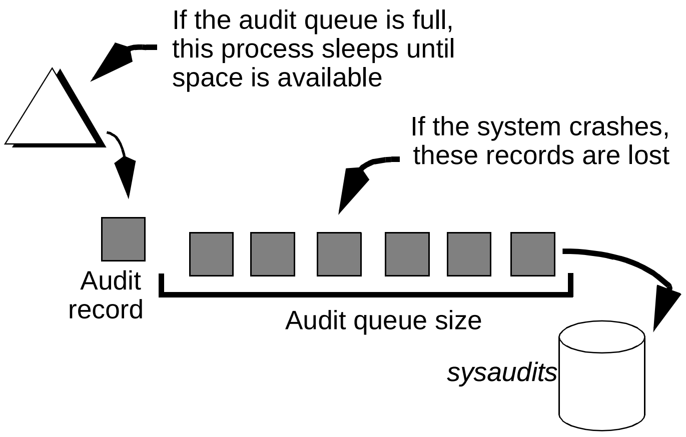
There are trade-offs in sizing the audit queue, as shown in Figure 5-5.

If the audit queue is large, so that you do not risk having user processes sleep, you run the risk of losing any audit records in memory if there is a system failure. The maximum number of records that can be lost is the maximum number of records that can be stored in the audit queue.

If security is your chief concern, keep the queue small. If you can risk the loss of more audit records, and you require high performance, make the queue larger.

Increasing the size of the in-memory audit queue takes memory from the total memory allocated to the data cache.

Figure 5-5: Trade-offs in auditing and performance


If the audit queue is full, this process sleeps until space is available
If the system crashes, these records are lost
Audit record
Audit queue size
sysaudits

Auditing performance guidelines

- Heavy auditing slows overall system performance. Audit only the events you need to track.
- If possible, place the sysaudits database on its own device. If that is impossible, place it on a device that is not used for your most critical applications.

Text and image pages

Text and image pages can use large portions of memory and are commonly known as space wasters. They exist as long as a parent data row points to the text and image pages. These pages come into existence when a null update is done against the columns.

Find the current status for the table:

```
sp_help table_name
```

Use sp_chcattribure to deallocate text and image pages to open the space they occupy:

```
sp_chgattribute table_name, "deallocate_first_txtpg",1
```

This switches the deallocation on. To switch the deallocation off enter:

```
sp_chgattribute table_name, “deallocate_first_txtpg”,0
```

CHAPTER 6 Tuning Asynchronous Prefetch

This chapter explains how asynchronous prefetch improves I/O performance for many types of queries by reading data and index pages into cache before they are needed by the query.

How asynchronous prefetch improves performance

Asynchronous prefetch improves performance by anticipating the pages required for certain well-defined classes of database activities for which access patterns are predictable. The I/O requests for these pages are issued before the query needs them so that most pages are in cache by the time query processing needs to access the page. Asynchronous prefetch can improve performance for:

- Sequential scans, such as table scans, clustered index scans, and covered nonclustered index scans
- Access via nonclustered indexes
- Some dbcc checks and update statistics
- Recovery

Asynchronous prefetch can improve the performance of queries that access large numbers of pages, such as decision-support applications, as long as the I/O subsystems on the machine are not saturated.

Asynchronous prefetch cannot help (or may help only slightly) when the I/O subsystem is already saturated or when Adaptive Server is CPU-bound. Asynchronous prefetch can be used in some OLTP applications, but to a much lesser degree, since OLTP queries generally perform fewer I/O operations.

When a query in Adaptive Server needs to perform a table scan, it:

- Examines the rows on a page and the values in the rows.
- Checks the cache for the next page to be read from a table. If that page is in cache, the task continues processing. If the page is not in cache, the task issues an I/O request and sleeps until the I/O completes.
- When the I/O completes, the task moves from the sleep queue to the run queue. When the task is scheduled on an engine, Adaptive Server examines rows on the newly fetched page.

This cycle of executing and stalling for disk reads continues until the table scan completes. In a similar way, queries that use a nonclustered index process a data page, issue the I/O for the next page referenced by the index, and sleep until the I/O completes, if the page is not in cache.

This pattern of executing and then waiting for I/O slows performance for queries that issue physical I/Os for large number of pages. In addition to the waiting time for the physical I/Os to complete, the task repeatedly switches on and off the engine, adding overhead to processing.

Improving query performance by prefetching pages

Asynchronous prefetch issues I/O requests for pages before the query needs them so that most pages are in cache by the time query processing needs to access the page. If required pages are already in cache, the query does not yield the engine to wait for the physical read. The query may still yield for other reasons, but it yields less frequently.

Based on the type of query being executed, asynchronous prefetch builds a **look-ahead set** of pages that it predicts will be needed very soon. Adaptive Server defines different look-ahead sets for each processing type where asynchronous prefetch is used.

In some cases, look-ahead sets are extremely precise; in others, some assumptions and speculation may lead to pages being fetched that are never read. When only a small percentage of unneeded pages are read into cache, the performance gains of asynchronous prefetch far outweigh the penalty for the wasted reads. If the number of unused pages becomes large, Adaptive Server detects this condition and either reduces the size of the look-ahead set or temporarily disables prefetching.

Prefetching control mechanisms in a multiuser environment

When many simultaneous queries are prefetching large numbers of pages into a buffer pool, there is a risk that the buffers fetched for one query could be flushed from the pool before they are used.

Adaptive Server tracks the buffers brought into each pool by asynchronous prefetch and the number that are used. It maintains a per-pool count of prefetched but unused buffers. By default, Adaptive Server sets an asynchronous prefetch limit of 10 percent of each pool. In addition, the limit on the number of prefetched but unused buffers is configurable on a per-pool basis.

The pool limits and usage statistics act like a governor on asynchronous prefetch to keep the cache-hit ratio high and reduce unneeded I/O. Overall, the effect ensures that most queries experience a high cache-hit ratio and few stalls due to disk I/O sleeps.

The following sections describe how the look-ahead set is constructed for the activities and query types that use asynchronous prefetch. In some asynchronous prefetch optimizations, allocation pages are used to build the look-ahead set.

For information on how allocation pages record information about object storage, see Chapter 2, “Data Storage,” in *Performance and Tuning Series: Physical Database Tuning*.

Look-ahead set during recovery

During recovery, Adaptive Server reads each log page that includes records for a transaction and then reads all the data and index pages referenced by that transaction, to verify timestamps and to roll transactions back or forward. Then, it performs the same work for the next completed transaction, until all transactions for a database have been processed. Two separate asynchronous prefetch activities speed recovery: asynchronous prefetch on the log pages themselves and asynchronous prefetch on the referenced data and index pages.

Prefetching log pages

The transaction log is stored sequentially on disk, filling extents in each allocation unit. Each time the recovery process reads a log page from a new allocation unit, it prefetches all the pages on that allocation unit that are in use by the log.

In databases that do not have a separate log segment, log and data extents may be mixed on the same allocation unit. Asynchronous prefetch still fetches all the log pages on the allocation unit, but the look-ahead sets may be smaller.

Prefetching data and index pages

For each transaction, Adaptive Server scans the log, building the look-ahead set from each referenced data and index page. While one transaction's log records are being processed, asynchronous prefetch issues requests for the data and index pages referenced by subsequent transactions in the log, reading the pages for transactions ahead of the current transaction.

Note Recovery uses only the pool in the default data cache. See "Setting limits for recovery" on page 144 for more information.

Look-ahead set during sequential scans

Sequential scans include table scans, clustered index scans, and covered nonclustered index scans.

During table scans and clustered index scans, asynchronous prefetch uses allocation page information about the pages used by the object to construct the look-ahead set. Each time a page is fetched from a new allocation unit, the look-ahead set is built from all the pages on that allocation unit that are used by the object.

The number of times a sequential scan hops between allocation units is kept to measure fragmentation of the page chain. This value is used to adapt the size of the look-ahead set so that large numbers of pages are prefetched when fragmentation is low, and smaller numbers of pages are fetched when fragmentation is high. See “Page chain fragmentation” on page 137.

Look-ahead set during nonclustered index access

When using a nonclustered index to access rows, asynchronous prefetch finds the page numbers for all qualified index values on a nonclustered index leaf page. It builds the look-ahead set from the unique list of all the pages that are needed.

Asynchronous prefetch is used only if two or more rows qualify.

If a nonclustered index access requires several leaf-level pages, asynchronous prefetch requests are also issued on the leaf pages.

Look-ahead set during *dbcc* checks

Asynchronous prefetch is used during the following dbcc checks:

- dbcc checkalloc, which checks allocation for all tables and indexes in a database, and the corresponding object-level commands, dbcc tablealloc and dbcc indexalloc
- dbcc checkdb, which checks all tables and index links in a database, and dbcc checktable, which checks individual tables and their indexes

Allocation checking

The dbcc commands checkalloc, tablealloc and indexalloc, which check page allocations, validate information on the allocation page. The look-ahead set for the dbcc operations that check allocation is similar to the look-ahead set for other sequential scans. When the scan enters a different allocation unit for the object, the look-ahead set is built from all the pages on the allocation unit that are used by the object.

checkdb and *checktable*

The dbcc checkdb and dbcc checktable commands check the page chains for a table, building the look-ahead set in the same way as other sequential scans.

If the table being checked has nonclustered indexes, the indexes are scanned recursively, starting at the root page and following all pointers to the data pages. When checking the pointers from the leaf pages to the data pages, the dbcc commands use asynchronous prefetch in a way that is similar to nonclustered index scans. When a leaf-level index page is accessed, the look-ahead set is built from the page IDs of all the pages referenced on the leaf-level index page.

Look-ahead set minimum and maximum sizes

The size of a look-ahead set for a query at a given point in time is determined by:

- The type of query, such as a sequential scan or a nonclustered index scan
- The size of the pools used by the objects that are referenced by the query and the prefetch limit set on each pool
- The fragmentation of tables or indexes, in the case of operations that perform scans
- The recent success rate of asynchronous prefetch requests and overload conditions on I/O queues and server I/O limits

Table 6-1 summarizes the minimum and maximum sizes for different type of asynchronous prefetch usage.

Table 6-1: Look-ahead set sizes

Access type	Action	Look-ahead set sizes
Table scan Clustered index scan Covered leaf-level scan	Reading a page from a new allocation unit	Minimum is eight pages needed by the query Maximum is the smaller of: • The number of pages on an allocation unit that belong to an object. • The pool prefetch limits
Nonclustered index scan	Locating qualified rows on the leaf page and preparing to access data pages	Minimum is two qualified rows Maximum is the smaller of: • The number of unique page numbers on qualified rows on the leaf index page • The pool's prefetch limit
Recovery	Recovering a transaction	Maximum is the smaller of: • All of the data and index pages touched by a transaction undergoing recovery • The prefetch limit of the pool in the default data cache
	Scanning the transaction log	Maximum is all pages on an allocation unit belonging to the log
dbcc tablealloc, indexalloc, and checkalloc	Scanning the page chain	Same as table scan
dbcc checktable and checkdb	Scanning the page chain	Same as table scan
	Checking nonclustered index links to data pages	All of the data pages referenced on a leaf-level page.

When prefetch is automatically disabled

Asynchronous prefetch attempts to fetch needed pages into buffer pools without flooding the pools or the I/O subsystem, and without reading unneeded pages. If Adaptive Server detects that prefetched pages are being read into cache but not used, it temporarily limits or discontinues asynchronous prefetch.

Flooding pools

For each pool in the data caches, a configurable percentage of buffers can be read in by asynchronous prefetch and held until the buffers' first use. For example, if a 2K pool has 4000 buffers, and the limit for the pool is 10 percent, then, at most, 400 buffers can be read in by asynchronous prefetch and remain unused in the pool. If the number of nonaccessed prefetched buffers in the pool reaches 400, Adaptive Server temporarily discontinues asynchronous prefetch for that pool.

As the pages in the pool are accessed by queries, the count of unused buffers in the pool drops, and asynchronous prefetch resumes operation. If the number of available buffers is smaller than the number of buffers in the look-ahead set, only that many asynchronous prefetches are issued. For example, if 350 unused buffers are in a pool that allows 400, and a query's look-ahead set is 100 pages, only the first 50 asynchronous prefetches are issued.

This keeps multiple asynchronous prefetch requests from flooding the pool with requests that flush pages out of cache before they can be read. The number of asynchronous I/Os that cannot be issued due to the per-pool limits is reported by sp_sysmon.

I/O system overloads

Adaptive Server and the operating system place limits on the number of outstanding I/Os for the server as a whole and for each engine. The configuration parameters max async i/os per server and max async i/os per engine control these limits for Adaptive Server.

See your operating system documentation for more information about configuring I/Os for your hardware.

The configuration parameter disk i/o structures controls the number of disk control blocks that Adaptive Server reserves. Each physical I/O (each buffer read or written) requires one control block while it is in the I/O queue.

See Chapter 5, "Setting Configuration Parameters," in the *System Administration Guide: Volume 1*.

If Adaptive Server tries to issue asynchronous prefetch requests that would exceed max async i/os per server, max async i/os per engine, or disk i/o structures, it issues enough requests to reach the limit and discards the remaining requests. For example, if only 50 disk I/O structures are available, and the server attempts to prefetch 80 pages, 50 requests are issued, and the other 30 are discarded.

sp_sysmon reports the number of times these limits are exceeded by asynchronous prefetch requests. See *Performance and Tuning Series: Monitoring Adaptive Server with sp_sysmon*.

Try to tune the system so there are no delayed I/Os. If there are I/Os delayed by:

- Disk I/O structures, increase the number of disk i/o structures configuration parameter
- The server or engine limit, increase the max max async i/os per engine and max async i/os per server configuration parameters.
- The operating system, tune the operating system so it can handle more concurrent I/Os.

Unnecessary reads

Asynchronous prefetch tries to avoid unnecessary physical reads. During recovery and during nonclustered index scans, look-ahead sets are exact, fetching only the pages referenced by page number in the transaction log or on index pages.

Look-ahead sets for table scans, clustered index scans, and dbcc checks are more speculative and may lead to unnecessary reads. During sequential scans, unnecessary I/O can take place due to:

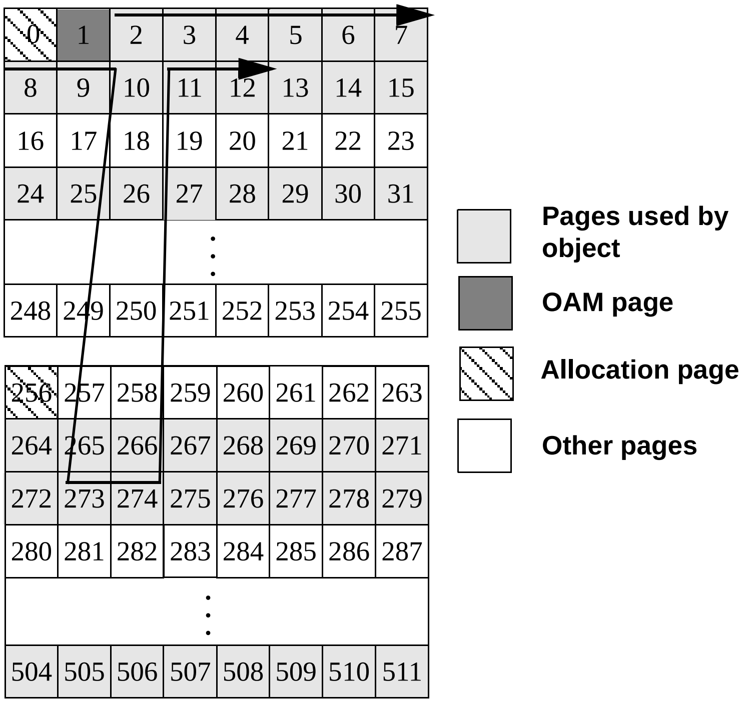
- Page chain fragmentation on allpages-locked tables
- Heavy cache utilization by multiple users

Page chain fragmentation

Adaptive Server page allocation mechanism strives to keep pages that belong to the same object close to each other in physical storage by allocating new pages on an extent already allocated to the object and by allocating new extents on allocation units already used by the object.

However, as pages are allocated and deallocated, page chains on data-only-locked tables can develop kinks. Figure 6-1 shows an example of a kinked page chain between extents in two allocation units.

Figure 6-1: A kink in a page chain crossing allocation units



In Figure 6-1, when a scan first needs to access a page from allocation unit 0, it checks the allocation page and issues asynchronous I/Os for all the pages used by the object it is scanning, up to the limit set on the pool. As the pages become available in cache, the query processes them in order by following the page chain. When the scan reaches page 10, the next page in the page chain, page 273, belongs to allocation unit 256.

When page 273 is needed, allocation page 256 is checked, and asynchronous prefetch requests are issued for all the pages in that allocation unit that belong to the object.

When the page chain points back to a page in allocation unit 0, there are two possibilities:

- The prefetched pages from allocation unit 0 are still in cache, and the query continues processing with no unneeded physical I/Os.
- The prefetch pages from allocation unit 0 have been flushed from the cache by the reads from allocation unit 256 and other I/Os taking place by other queries that use the pool. The query must reissue the prefetch requests. This condition is detected in two ways:
    - Adaptive Server's count of the hops between allocation pages now equals two. Adaptive Server uses the ratio between the count of hops and the prefetched pages to reduce the size of the look-ahead set, so fewer I/Os are issued.
    - The count of prefetched but unused pages in the pool is likely to be high, so asynchronous prefetch may be temporarily discontinued or reduced, based on the pool's limit.

Tuning goals for asynchronous prefetch

Choosing optimal pool sizes and prefetch percentages for buffer pools can be key to achieving improved performance with asynchronous prefetch. When multiple applications are running concurrently, a well-tuned prefetching system balances pool sizes and prefetch limits to accomplish:

- Improved system throughput
- Better performance by applications that use asynchronous prefetch
- No performance degradation in applications that do not use asynchronous prefetch

Configuration changes to pool sizes and the prefetch limits for pools are dynamic, allowing you to make changes to meet the needs of varying workloads. For example, you can configure asynchronous prefetch for good performance during recovery or dbcc checking and reconfigure afterward without needing to restart Adaptive Server.

See "Setting limits for recovery" on page 144 and "Setting limits for dbcc" on page 144.

Commands for configuration

Asynchronous prefetch limits are configured as a percentage of the pool in which prefetched but unused pages can be stored. There are two configuration levels:

- The server-wide default, set with the configuration parameter global async prefetch limit. When you install Adaptive Server, the default value for global async prefetch limit is 10 (percent).
- A per-pool override, set with sp_poolconfig. To see the limits set for each pool, use sp_cacheconfig.

See Chapter 5, "Setting Configuration Parameters," in the *System Administration Guide: Volume1*.

Changing asynchronous prefetch limits takes effect immediately, and does not require a restart. You can configure the global and per-pool limits in the configuration file.

Other Adaptive Server performance features

This section covers the interaction of asynchronous prefetch with other Adaptive Server performance features.

Large I/O

The combination of large I/O and asynchronous prefetch can provide rapid query processing with low I/O overhead for queries performing table scans and for dbcc operations.

When large I/O prefetches all the pages on an allocation unit, the minimum number of I/Os for the entire allocation unit is:

- 31 16K I/Os
- 7 2K I/Os, for the pages that share an extent with the allocation page

Note Reference to Large I/Os are on a 2K logical page size server. If you have an 8K page size server, the basic unit for the I/O is 8K. If you have a 16K page size server, the basic unit for the I/O is 16K.

Sizing and limits for the 16K pool

Performing 31 16K prefetches with the default asynchronous prefetch limit of 10 percent of the buffers in the pool requires a pool with at least 310 16K buffers. If the pool is smaller, or if the limit is lower, some prefetch requests will be denied. To allow more asynchronous prefetch activity in the pool, configure either a larger pool or a larger prefetch limit for the pool.

If multiple overlapping queries perform table scans using the same pool, the number of unused, prefetched pages allowed in the pool needs to be higher. The queries are probably issuing prefetch requests at slightly staggered times and are at different stages in reading the accessed pages. For example, one query may have just prefetched 31 pages, and have 31 unused pages in the pool, while an earlier query has only 2 or 3 unused pages left. To start your tuning efforts for these queries, assume one-half the number of pages for a prefetch request multiplied by the number of active queries in the pool.

Limits for the 2K pool

Queries using large I/O during sequential scans may still need to perform 2K I/O:

- When a scan enters a new allocation unit, it performs 2K I/O on the 7 pages in the unit that share space with the allocation page.
- If pages from the allocation unit already reside in the 2K pool when the prefetch requests are issued, the pages that share that extent must be read into the 2K pool.

If the 2K pool has its asynchronous prefetch limit set to 0, the first 7 reads are performed by normal asynchronous I/O, and the query sleeps on each read if the pages are not in cache. Set the limits on the 2K pool high enough that it does not slow prefetching performance.

Fetch-and-discard (MRU) scans

When a scan uses MRU replacement policy, buffers are handled in a special manner when they are read into the cache by asynchronous prefetch. First, pages are linked at the MRU end of the chain, rather than at the wash marker. When the query accesses the page, the buffers are relinked into the pool at the wash marker. This strategy helps to avoid cases where heavy use of a cache flushes prefetched buffers linked at the wash marker before the buffers can be used. It has little impact on performance, unless large numbers of unneeded pages are being prefetched. In this case, the prefetched pages are more likely to flush other pages from cache.

Parallel scans and large I/Os

The demand on buffer pools can become higher with parallel queries. With serial queries operating on the same pools, it is safe to assume that queries are issued at slightly different times and that the queries are in different stages of execution: some are accessing pages are already in cache, and others are waiting on I/O.

Parallel execution places different demands on buffer pools, depending on the type of scan and the degree of parallelism. Some parallel queries are likely to issue a large number of prefetch requests simultaneously.

Hash-based table scans

Hash-based table scans on allpages-locked tables have multiple worker processes that all access the same page chain. Each worker process checks the page ID of each page in the table, but examines only the rows on those pages where page ID matches the hash value for the worker process.

The first worker process that needs a page from a new allocation unit issues a prefetch request for all pages from that unit. When the scans of other worker processes also need pages from that allocation unit, the scans will either find that the pages they need are already in I/O or already in cache. As the first scan to complete enters the next unit, the process is repeated.

As long as one worker process in the family performing a hash-based scan does not become stalled (waiting for a lock, for example), the hash-based table scans do not place higher demands on the pools than they place on serial processes. Since the multiple processes may read the pages much more quickly than a serial process does, they change the status of the pages from unused to used more quickly.

Partition-based scans

Partition-based scans are more likely to create additional demands on pools, since multiple worker processes may be performing asynchronous prefetching on different allocation units. On partitioned tables on multiple devices, the per-server and per-engine I/O limits are less likely to be reached, but the per-pool limits are more likely to limit prefetching.

Once a parallel query is parsed and compiled, it launches its worker processes. If a table with 4 partitions is being scanned by 4 worker processes, each worker process attempts to prefetch all the pages in its first allocation unit. For the performance of this single query, the most desirable outcome is that the size and limits on the 16K pool are sufficiently large to allow 124 (31*4) asynchronous prefetch requests, so all of the requests succeed. Each of the worker processes scans the pages in cache quickly, moving onto new allocation units and issuing more prefetch requests for large numbers of pages.

Special settings for asynchronous prefetch limits

You may want to change asynchronous prefetch configuration temporarily for specific purposes, including:

- Recovery
- `dbcc` operations that use asynchronous prefetch

Setting limits for recovery

During recovery, Adaptive Server uses only the 2K pool of the default data cache. If you shut down the server using shutdown with nowait, or if the server goes down due to power failure or machine failure, the number of log records to be recovered may be quite large.

To speed recovery, edit the configuration file to do one or both of the following:

- Increase the size of the 2K pool in the default data cache by reducing the size of other pools in the cache
- Increase the prefetch limit for the 2K pool

Both of these configuration changes are dynamic, so you can use sp_poolconfig to restore the original values after recovery completes, without restarting Adaptive Server. The recovery process allows users to log in to the server as soon as recovery of the master database is complete. Databases are recovered one at a time and users can begin using a particular database as soon as it is recovered. There may be some contention if recovery is still taking place on some databases, and user activity in the 2K pool of the default data cache is heavy.

Setting limits for *dbcc*

If you are performing database consistency checking when other activity on the server is low, configuring high asynchronous prefetch limits on the pools used by dbcc can speed consistency checking.

dbcc checkalloc can use special internal 16K buffers if there is no 16K pool in the cache for the appropriate database. If you have a 2K pool for a database, and no 16K pool, set the local prefetch limit to 0 for the pool while executing dbcc checkalloc. Use of the 2K pool instead of the 16K internal buffers may actually hurt performance.

Maintenance activities for high prefetch performance

Page chains for all pages-locked tables and the leaf levels of indexes develop kinks as data modifications take place on the table. In general, newly created tables have few kinks. Tables where updates, deletes, and inserts that have caused page splits, new page allocations, and page deallocations are likely to have cross-allocation unit page chain kinks. If more than 10 to 20 percent of the original rows in a table have been modified, determine if kinked page chains are reducing the effectiveness of asynchronous prefetch. If you suspect that page chain kinks are reducing asynchronous prefetch performance, you may need to re-create indexes or reload tables to reduce kinks.

Eliminating kinks in heap tables

For allpages-locked heaps, page allocation is generally sequential, unless pages are deallocated by deletions that remove all rows from a page. These pages may be reused when additional space is allocated to the object. You can create a clustered index (and drop it, if you want the table stored as a heap) or bulk copy the data out, truncate the table, and copy the data in again. Both activities compress the space used by the table and eliminate page-chain kinks.

Eliminating kinks in clustered index tables

For clustered indexes, page splits and page deallocations can cause page chain kinks. Rebuilding clustered indexes does not necessarily eliminate all cross-allocation page linkages. Use fillfactor for clustered indexes where you expect growth, to reduce the number of kinks resulting from data modifications.

Eliminating kinks in nonclustered indexes

If your query mix uses covered index scans, dropping and re-creating nonclustered indexes can improve asynchronous prefetch performance, once the leaf-level page chain becomes fragmented.

Performance monitoring and asynchronous prefetch

The output of statistics io reports the number physical reads performed by asynchronous prefetch and the number of reads performed by normal asynchronous I/O. In addition, statistics io reports the number of times that a search for a page in cache was found by the asynchronous prefetch without holding the cache spinlock.

See Chapter 1, Using the set statistics Commands" of the *Performance and Tuning Series: Improving Performance with Statistical Analysis*.

The sp_sysmon report contains information on asynchronous prefetch in both the "Data Cache Management" section and the "Disk I/O Management" section.

If you use sp_sysmon to evaluate asynchronous prefetch performance, you may see improvements in other performance areas, such as:

- Much higher cache hit ratios in the pools where asynchronous prefetch is effective.
- A corresponding reduction in context switches due to cache misses, with voluntary yields increasing.
- A possible reduction in lock contention. Tasks keep pages locked during the time it takes for perform I/O for the next page needed by the query. If this time is reduced because asynchronous prefetch increases cache hits, locks are held for a shorter time.

See *Performance and Tuning Series: Monitoring Adaptive Server with sp_sysmon*.

Index