Performance and Tuning Series: Physical Database Tuning

Adaptive Server® Enterprise

15.7

DOCUMENT ID: DC00841-01-1570-01

LAST REVISED: September 2011

Copyright © 2011 by Sybase, Inc. All rights reserved.

This publication pertains to Sybase software and to any subsequent release until otherwise indicated in new editions or technical notes. Information in this document is subject to change without notice. The software described herein is furnished under a license agreement, and it may be used or copied only in accordance with the terms of that agreement.

To order additional documents, U.S. and Canadian customers should call Customer Fulfillment at (800) 685-8225, fax (617) 229-9845.

Customers in other countries with a U.S. license agreement may contact Customer Fulfillment via the above fax number. All other international customers should contact their Sybase subsidiary or local distributor. Upgrades are provided only at regularly scheduled software release dates. No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of Sybase, Inc.

Sybase trademarks can be viewed at the Sybase trademarks page at http://www.sybase.com/detail?id=1011207. Sybase and the marks listed are trademarks of Sybase, Inc. ® indicates registration in the United States of America.

SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Unicode and the Unicode Logo are registered trademarks of Unicode, Inc.

IBM and Tivoli are registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

All other company and product names mentioned may be trademarks of the respective companies with which they are associated.

Use, duplication, or disclosure by the government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of DFARS 52.227-7013 for the DOD and as set forth in FAR 52.227-19(a)-(d) for civilian agencies.

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

Contents

CHAPTER 1 **Controlling Physical Data Placement** **1**

Improving performance by controling object placement 2

Identifying poor object placement 3

Using sp_sysmon while changing data placement 4

Improving I/O performance 4

Spreading data across disks to avoid I/O contention 4

Isolating server-wide I/O from database I/O 5

Keeping transaction logs on a separate disk 6

Mirroring a device on a separate disk 7

Using segments 8

Creating objects on segments 9

Separating tables and indexes 10

Splitting large tables across devices 10

Moving text storage to a separate device 10

Partitioning tables for performance 11

How Adaptive Server distributes partitions on devices 11

Space planning for partitioned tables 12

Read-only tables 13

Read-mostly tables 13

Tables with random data modification 14

Adding disks when devices are full 15

Adding disks when devices are full 15

Adding disks when devices are nearly full 16

Maintenance issues and partitioned tables 18

Regular maintenance checks for partitioned tables 18

CHAPTER 2 **Data Storage** **21**

Query optimization 21

Query processing and page reads 22

Adaptive Server pages 23

Page headers and page sizes 24

Data and index pages 24

Large object (LOB) pages 25

Extents 25

Pages that manage space allocation 26
Global allocation map pages 27
Allocation pages 27
Object allocation map pages 27
How OAM pages and allocation pages manage object storage 28
Page allocation keeps an object's pages together 29
Data access using sysindexes and syspartitions 29
Space overheads 30
Number of columns and size 31
Number of rows per data page 37
Additional number of object and size restrictions 38
Tables without clustered indexes 38
Locking schemes 39
Select operations on heap tables 40
Inserting data into an allpages-locked heap table 41
Inserting data into a data-only-locked heap table 42
Deleting data from a heap table 43
Updating data on a heap table 44
How Adaptive Server performs I/O for heap operations 45
Maintaining heap tables 46
Transaction log: a special heap table 47
Asynchronous prefetch and I/O on heap tables 48
Caches and object bindings 49
Heap tables, I/O, and cache strategies 49
Select operations and caching 51
Data modification and caching 51

CHAPTER 3 Setting Space Management Properties 55
Reducing index maintenance 55
Advantages of using fillfactor 56
Disadvantages of using fillfactor 57
Setting fillfactor values 57
fillfactor examples 58
Using the sorted_data and fillfactor options 61
Reducing row forwarding 62
Default, minimum, and maximum values for exp_row_size 62
Specifying an expected row size with create table 63
Adding or changing an expected row size 64
Setting a default expected row size server-wide 64
Displaying the expected row size for a table 65
Choosing an expected row size for a table 65
Conversion of max_rows_per_page to exp_row_size 66
Monitoring and managing tables that use expected row size .. 67
Leaving space for forwarded rows and inserts 68

Extent allocation commands and reservepagegap 68
Specifying a reserve page gap with create table..................... 69
Specifying a reserve page gap with create index..................... 70
Changing reservepagegap 70
reservepagegap examples 71
Choosing a value for reservepagegap 72
Monitoring reservepagegap settings 73
reservepagegap and sorted_data options................................ 73
Using max_rows_per_page on allpages-locked tables.................. 75
Reducing lock contention 76
Indexes and max_rows_per_page 77
select into and max_rows_per_page....................................... 77
Applying max_rows_per_page to existing data........................ 77

CHAPTER 4 Table and Index Size.. 79
Determining the sizes of tables and indexes.................................. 80
Effects of data modifications on object sizes 81
Using optdiag to display object sizes 81
Advantages of optdiag.. 82
Disadvantages of optdiag.. 82
Using sp_spaceused to display object size.................................... 82
Advantages of sp_spaceused 83
Disadvantages of sp_spaceused 84
Using sp_estspace to estimate object size 84
Advantages of sp_estspace 86
Disadvantages of sp_estspace 86
Using formulas to estimate object size... 86
Factors that can affect storage size 87
Storage sizes for datatypes... 87
Tables and indexes used in the formulas................................ 89
Calculating table and clustered index sizes for allpages-locked tables 89
Calculating the sizes of data-only-locked tables 96
Other factors affecting object size 101
Very small rows 102
LOB pages 103
Advantages of using formulas to estimate object size 104
Disadvantages of using formulas to estimate object size...... 104

CHAPTER 5 Database Maintenance 105
Running reorg on tables and indexes 105
Creating and maintaining indexes.. 106
Configuring Adaptive Server to speed sorting....................... 106

Dumping the database after creating an index 107
Creating an index on sorted data 107
Maintaining index and column statistics 108
Rebuilding indexes 109
Creating or altering a database 110
Backup and recovery 112
Local backups 112
Remote backups 112
Online backups 112
Using thresholds to prevent running out of log space 113
Minimizing recovery time 113
Recovery order 113
Bulk-copy 113
Parallel bulk-copy 114
Batches and bulk-copy 114
Slow bulk-copy 115
Improving bulk-copy performance 115
Replacing the data in a large table 116
Adding large amounts of data to a table 116
Using partitions and multiple bulk-copy processes 116
Impacts on other users 116
Database consistency checker 117
Using dbcc tune (cleanup) 117
Using dbcc tune on spinlocks 117
Determining the space available for maintenance activities 118
Overview of space requirements 119
Checking space usage and space available 119
Estimating the effects of space management properties 122
If there is not enough space 123

CHAPTER 6 **Temporary Databases 125**
How temporary database management affects performance 125
Using temporary tables 126
Hashed (#) temporary tables 126
Regular user tables 127
Worktables 128
Temporary databases 128
Session-assigned temporary database 128
Using multiple temporary databases 129
Creating user temporary databases 129
Configuring a default tempdb group 130
Binding to groups and tempdb 130
Tuning system temporary databases for performance 131
Placing system tempdb 131

Configuring user-created temporary databases 134
General guidelines 134
Logging optimizations for temporary databases 140
User log cache (ULC).. 141

Index 143

CHAPTER 1 Controlling Physical Data Placement

This chapter explains how to improve performance by controlling the location of tables and indexes.

Topic	Page
Improving performance by controling object placement	2
Improving I/O performance	4
Partitioning tables for performance	11
Space planning for partitioned tables	12
Adding disks when devices are full	15
Maintenance issues and partitioned tables	18

To make the most of physical database tuning, understand these distinctions between logical and physical devices:

- The physical disk or physical device is the hardware that stores the data.
- A database device or logical device is all or part of7 a physical disk that has been initialized (with the disk init command) for use by Adaptive Server®. A database device can be an operating system file, an entire disk, or a disk partition.

  See the *Installation Guide* and the *Configuration Guide* for your platform for information about specific operating system constraints on disk and file usage.
- A segment is a named collection of database devices used by a database. The database devices that make up a segment can be located on separate physical devices.

- A partition is a subset of a table. Partitions are database objects that can be managed independently. You can split partitioned tables, so multiple tasks can access it simultaneously. You can place a partition on a specific segment. If each partition is on a different segment and each segment has its own database device, queries accessing these tables benefit from improved parallelism. See create table in the *Reference Manual: Commands* and the *Transact-SQL Users Guide* for more information about creating and using partitions.

Use sp_helpdevice, sp_helpsegment, and sp_helpartition to get more information about devices, segments, and partitions.

Improving performance by controling object placement

Adaptive Server allows you to control the placement of databases, tables, and indexes across physical storage devices, which can improve performance by equalizing the reads and writes to disk across many devices and controllers. For example, you can:

- Place database data segments on a specific device or devices, storing the database log on a separate physical device so that reads and writes to the log do not interfere with data access.
- Spread large, heavily used tables across several devices.
- Place specific tables or nonclustered indexes on specific devices. For example, you might place a table on a segment that spans several devices and its nonclustered indexes on a separate segment.
- Place the text and image page chain for a table on a separate device from the table. The table stores a pointer to the actual data value in the separate database structure, so each access to a text or image column requires at least two I/Os.
- Distribute tables evenly across partitions on separate physical disks to provide optimum parallel query performance and improve insert and update performance.

For multiuser and multi-CPU systems that perform a lot of disk I/O, be especially aware of physical and logical device issues and the distribution of I/O across devices:

- Plan a balanced separation of objects across logical and physical devices.

- Use enough physical devices, including disk controllers, to ensure physical bandwidth.
- Use an increased number of logical devices to ensure minimal contention for internal I/O queues.
- Determine and use a number of partitions that allows parallel scans and meets query performance goals.

Identifying poor object placement

Your system may benefit from more appropriately placed objects if:

- Single-user performance is satisfactory, but response time increases significantly when Adaptive Server executes multiple processes.
- Access to a mirrored disk takes twice as long as access to an unmirrored disk.
- Objects that are frequently accessed ("hot objects") degrade the performance of queries that use the tables in which these objects are located.
- Maintenance activities take a long time.
- tempdb performance is affected if it shares disk space with other databases. Most system procedures and applications use tempdb as their workspace, and are adversely affected if tempdb shares the same disk with other databases.
- insert performance is poor on heavily used tables.
- Queries that run in parallel perform poorly, due to an imbalance of data pages on partitions or devices, or they run in serial, due to extreme imbalance.

If you experience problems due to disk contention and other problems related to object placement, check for and correct these issues:

- Random-access (I/O for data and indexes) and serial-access (log I/O) processes use the same disks.
- Database processes and operating system processes use the same disks.
- Serial disk mirroring.
- Database logging or auditing takes place on the same disk as data storage.

Using *sp_sysmon* while changing data placement

Use sp_sysmon to determine whether data placement across physical devices is causing performance problems. Check the entire sp_sysmon output during tuning to verify how the changes affect all performance categories.

Pay special attention to the output associated with:

- I/O device contentions
- All-pages locked heap tables
- Last page locks on heaps
- Disk I/O management

See *Monitoring Adaptive Server with sp_sysmon*.

Improving I/O performance

To improve I/O performance in Adaptive Server, try:

- Spreading data across disks to avoid I/O contention
- Isolating server-wide I/O from database I/O
- Separating data storage and log storage for frequently updated databases
- Keeping random disk I/O away from sequential disk I/O
- Mirroring devices on separate physical disks
- Using partitions to distribute table data across devices

Spreading data across disks to avoid I/O contention

Avoid bottlenecks by spreading data storage across multiple disks and multiple disk controllers.

- Place databases with critical performance requirements on separate devices. If possible, also use separate controllers than those used by other databases. Use segments as needed for critical tables, and partitions as needed for parallel queries.
- Put heavily used and frequently joined tables on separate disks.

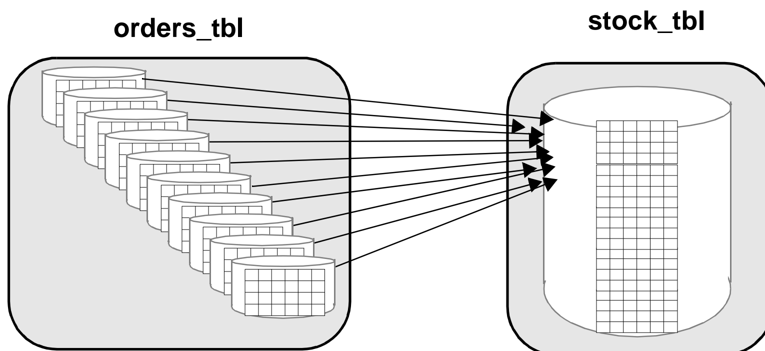
- Use segments to place tables and indexes on their own disks.

Avoiding physical contention in parallel join queries

Figure 1-1 illustrates a join of two tables, orders_tbl and stock_tbl. There are 10 worker process available: orders_tbl has 10 partitions on 10 different physical devices and is the outer table in the join; stock_tbl is nonpartitioned. The worker processes have a problem with access contention on orders_tbl, but each worker process must scan stock_tbl. There may be physical I/O contention if the entire table does not fit into cache. In the worst case, 10 worker processes attempt to access the physical device on which stock_tbl resides. Avoid physical I/O contention by creating a named cache that contains the entire table stock_tbl.

Another way to reduce or eliminate physical I/O contention is to partition both orders_tbl and stock_tbl and distribute those partitions on different physical devices.

Figure 1-1: Joining tables on different physical devices



Isolating server-wide I/O from database I/O

Place system databases with heavy I/O requirements (for example, tempdb and sybsecurity) on physical disks and controllers other than where application databases reside.

tempdb

It is a heavily used database that affects all processes on the server and is used by most system procedures. It is automatically installed on the master device. If more space is needed, you can expand tempdb to other devices. If you expect tempdb to be quite active, place it on a disk—the fastest available—that is not used for other important database activity. .

On some UNIX systems, I/O to operating system files is significantly faster than I/O to raw devices. tempdb is always re-created, rather than recovered, after a shutdown; therefore, you may be able to improve performance by moving tempdb onto an operating system file instead of a raw device. Test this on your own system.

See Chapter 6, “Temporary Databases,” for more placement advice for tempdb.

sybsecurity

Once enabled, the auditing system performs frequent I/O to the sysaudits table in the sybsecurity database. If your applications perform a significant amount of auditing, place sybsecurity on a disk that is used for tables where fast response time is not critical. Ideally, place sybsecurity on its own device.

Use the threshold manager to monitor free space to avoid suspending user transactions if the audit database fills up. See Chapter 16, “Managing Free Space with Thresholds,” In *System Administration Guide, Volume 2* for information about determing appropriate thresholds.

Keeping transaction logs on a separate disk

Place transaction logs on a separate segment, preventing the logs from competing with other objects for disk space. Placing the logs on a separate physical disk:

- Improves performance by reducing I/O contention
- Ensures full recovery in the event of hard disk failures on the data device
- Speeds recovery, since simultaneous asynchronous prefetch requests can read ahead on both the log device and the data device without contention

Both create database and alter database require you to use with override before you can place the transaction log on the same device as the data.

The log device can experience significant I/O on systems with heavy update activity. Adaptive Server writes log pages to disk when transactions commit, and may need to read log pages into memory to replace deferred updates with deferred operations.

When log and data are on the same database devices, the extents allocated to store log pages are not contiguous; log extents and data extents are mixed. When the log is on its own device, Adaptive Server allocates the extents sequentially, thus reducing disk head travel and seeks, and maintaining a higher I/O rate.

Adaptive Server buffers log records for each user in a user log cache, which reduces contention for writing to the log page in memory. If log and data are on the same devices, user log cache buffering is disabled, which results in serious performance degradation on SMP systems.

See Chapter 6, “Overview of Disk Resource Issues,” in the *System Administration Guide: Volume 1*.

Mirroring a device on a separate disk

Disk mirroring is a high availability feature that allows Adaptive Server to duplicate the contents of an entire database device.

See Chapter 2, “Disk Mirroring,” in the *System Administration Guide, Volume 2*.

If you mirror data, put the mirror on a separate physical disk from the device that it mirrors, minimizing mirroring’s performance impact. Disk hardware failure often results in whole physical disks being lost or unavailable

If you do not use mirroring, or use operating system mirroring, you may see slight performance improvements by setting disable disk mirroring configuration paramter to 1.

Mirroring can increase the time taken to complete disk writes, since the writes are executed on both disks, either serially or simultaneously. Disk mirroring has no effect on the time required to read data.

Mirrored devices use one of two modes for disk writes:

- Nonserial mode – can require more time to complete a write than an unmirrored write requires. In nonserial mode, both writes start at the same time, and Adaptive Server waits for both to complete. The time to complete nonserial writes is the greater of the two I/O times.

- Serial mode – increases the time required to write data even more than nonserial mode. Adaptive Server starts the first write and waits for it to complete before starting the second write. The time required is the sum of the two I/O times.

Using serial mode

Despite its performance impact, serial mode is the default mode because it guards against failures that occur while a write is taking place.

Since serial mode waits until the first write is complete before starting the second write, a single failure cannot affect both disks. Using nonserial mode improves performance, but you risk losing data if a failure occurs that affects both writes.

Warning! If your mirrored database system must be absolutely reliable, use serial mode.

Using segments

A segment is a label that points to one or more logical devices. Use segments to improve throughput by:

- Splitting large tables across disks, including tables that are partitioned for parallel query performance
- Separating tables and their nonclustered indexes across disks
- Separating table partitions and index across the disks
- Placing the text and image page chain on a disk other than the one on which the table resides, where the pointers to the text values are stored

In addition, you can use segments to control space usage:

- Tables or partitions cannot grow larger than their segment allocation. You can use segments to limit the table or partition size.
- Tables or partitions on other segments cannot use the space allocated to objects on another segment.
- The threshold manager monitors space usage.

Creating objects on segments

Each database can use up to 32 segments, including the 3 segments that are created by the system (system, log segment, and default) when a database is created.

Tables and indexes are stored on segments. If you execute create table or create index without specifying a segment, the objects are stored on the default segment for the database. Naming a segment in either of these commands creates the object on that segment. You can use the sp_placeobject system procedure to assign all future space allocations to take place on a specified segment, so tables can span multiple segments.

A system administrator must initialize the device with disk init and allocate the device to the database. Alternatively, the database owner can do this using create database or alter database.

Once the devices are available to the database, the database owner or object owners can create segments and place objects on the devices.

When you create a user-defined segment, you can place tables, indexes, and partitions on that segment using the create table or create index commands:

```
create table tableA(...) on seg1
create nonclustered index myix on tableB(...)
      on seg2
```

This example creates the table fictionsales, which is partitioned by range according to values in the date column:

```
create table fictionsales
(store_id int not null,
order_num int not null,
date datetime not null)
partition by range (date)
(q1 values <= ("3/31/2005") on seg1,
q2 values <= ("6/30/2005") on seg2,
q3 values <= ("9/30/2005") on seg3,
q4 values <= ("12/31/2005") on seg4)
```

By controlling the location of critical tables, you can arrange for these tables and indexes to be spread across disks.

Separating tables and indexes

Use segments to place tables on one set of disks and nonclustered indexes on another set of disks. You cannot place a clustered index on a different segment than its data pages. When you create a clustered index using the on *segment_name* clause, the entire table is moved to the specified segment, and the clustered index tree is built on that segment.

You can improve performance by placing nonclustered indexes on a separate segment.

Splitting large tables across devices

Segments can span multiple devices, so you can use them to spread data across one or more disks. This can help balance the I/O load for large and busy tables. For parallel queries, it is essential that you create segments across multiple devices for I/O parallelism during partitioned-based scans.

See Chapter 8, "Creating and Using Segments," in the *System Administration Guide, Volume 2*.

Moving text storage to a separate device

When a table includes a text, image, or Java off-row datatype, the table itself stores a pointer to the data value. The actual data is stored on a separate linked list of pages called a large object chain (LOB).

Writing or reading a LOB value requires at least two disk accesses, one to read or write the pointer, and one for subsequent reads or writes for the data. If your application frequently reads or writes LOB values, you can improve performance by placing the LOB chain on a separate physical device. Isolate LOB chains on disks that are not busy with other application-related table or index access.

When you create a table with LOB columns, Adaptive Server creates a row in sysindexes and syspartitions for the object that stores the LOB data. The value in the name column is the table name prefixed with a "t"; the indid is always 255. If you have multiple LOB columns in a single table, there is only one object used to store the data. By default, this object is placed on the same segment as the table.

Use sp_placeobject to move all future allocations for the LOB columns to a separate segment.

Partitioning tables for performance

Partitioning a table can improve performance for several types of processes.

- Partitioning allows parallel query processing to access each partition of the table. Each worker process in a partitioned-based scan reads a separate partition.
- Partitioning allows you to load a table in parallel with bulk copy.

  For more information on parallel bcp, see the *Utility Programs* manual.
- Partitioning allows tou to distribute a table's I/O over multiple database devices.
- Semantic partitioning (range-, hash- and list-partitioned tables) improves response time because the query processor eliminates some partitions.
- Partitioning provides multiple insertion points for a heap table.

The tables you choose to partition and the type of partition depend on the performance issues you encounter and the performance goals for the queries on the tables.

See Chapter 10, "Partitioning Tables and Indexes" in the *Transact-SQL Users Guide* book for more information about, and examples using and creating partitions.

How Adaptive Server distributes partitions on devices

In versions earlier than 15.0, Adaptive Server automatically maintained an affinity between partitions and devices when you created multiple partitions on a segment that was mapped to multiple database devices. This is no longer the case in Adaptive Server 15.0 and later; all partitions are created on the first device. To achieve affinity between partitions and devices:

1. Create a segment for a particular device.
2. Explicitly place a partition on that segment.

You can create as many as 29 user segments, and you must use the alter table syntax from Adaptive Server version 15.0 and later to create the segments, because the earlier syntax (alter table t partition 20) does not support explicit placement of partitions on segments.

Achieve the best I/O performance for parallel queries by matching the number of partitions to the number of devices in the segment.

You can partition tables that use the text, image, or Java off-row data types. However, the columns themselves are not partitioned—they remain on a single page chain.

RAID devices and partitioned tables

A striped redundant array of independent disks (RAID) device can contain multiple physical disks, but Adaptive Server treats such a device as a single logical device. You can use multiple partitions on the single logical device and achieve good parallel query performance.

To determine the optimum number of partitions for your application mix, start with one partition for each device in the stripe set. Use your operating system utilities (vmstat, sar, and iostat on UNIX; Performance Monitor on Windows) to check utilization and latency.

To check maximum device throughput, use select count(*), using the index *table_name* clause to force a table scan if a nonclustered index exists. This command requires minimal CPU effort and creates very little contention for other resources.

Space planning for partitioned tables

When you are planning for partitioned tables, consider how to maintain:

- Load balance across the disk for partition-based scan performance and for I/O parallelism
- Cclustered indexes, which require approximately 120% of the space occupied by the table to drop and re-create the index or to run reorg rebuild

The space planning decisions you make depend on the:

- Availability of disk resources for storing tables
- Nature of your application mix and of the incoming data (for semantic-partitioned tables)

Estimate the frequency with which your partitioned tables need maintenance: some applications need indexes to be re-created frequently to maintain balance, while others need little maintenance.

For those applications that need frequent load balancing for performance, having space in which to re-create a clustered index or run reorg rebuild provides fastest and easiest results. However, since creating clustered indexes requires copying the data pages, the space available on the segment must be equal to approximately 120% of the space occupied by the table.

See "Determining the space available for maintenance activities" on page 118.

The following descriptions of read-only, read-mostly, and random data modification provide a general picture of the issues involved in object placement and in maintaining partitioned tables.

See Chapter 10, "Partitioning Tables and Indexes" in the *Transact-SQL Users Guide* for information about the specific tasks required during maintenance.

Read-only tables

Tables that are read-only, or that are rarely changed, can completely fill the space available on a segment, and do not require maintenance. If a table does not require a clustered index, you can use parallel bulk copy (parallel bcp) to completely fill the space on the segment.

If a clustered index is needed, the table's data pages can occupy up to 80% of the space in the segment. The clustered index tree requires about 20% of the space used by the table.

This space requirement varies, depending on the length of the key. Initially, loading the data into the table and creating the clustered index requires several steps, but once you have performed these steps, maintenance is minimal.

Read-mostly tables

The guidelines above for read-only tables also apply to read-mostly tables with very few inserts. The only exceptions are as follows:

- If there are inserts to the table, and the clustered index key does not balance new space allocations evenly across the partitions, the disks underlying some partitions may become full, and new extent allocations are made to a different physical disk. This process is called extent stealing.

In huge tables spread across many disks, a small percentage of allocations to other devices is not a problem. Detect extent stealing by using sp_helpsegment to check for devices that have no space available, and by using sp_helpartition to check for partitions that have disproportionate numbers of pages.

If the imbalance in partition size leads to degradation in parallel query response times or optimization, you may want to balance the distribution by using one of the methods described in Chapter 10, “Partitioning Tables and Indexes” in the *Transact-SQL Users Guide*.

- If the table is a heap, round-robin-partitioned table, the random nature of heap table inserts should keep partitions balanced.

  Take care with large bulk copy in operations. However, if the table is a semantic partitioned table, consider changing the partition condition using alter table... partition by for appropriate load balance.

  You can use parallel bulk copy (parallel bcp) to send rows to the partition with the smallest number of pages to balance the data across the partitions. See Chapter 4, “Using bcp to Transfer Data to and from Adaptive Server,” in the *Utility Guide*.

Tables with random data modification

Tables with clustered indexes that experience many inserts, updates, and deletes over time tend to lead to data pages that are approximately 70 to 75% full. This can lead to performance degradation in several ways:

- More pages must be read to access a given number of rows, requiring additional I/O and wasting data cache space.
- On tables that use allpages locking, the performance of large I/O and asynchronous prefetch suffers because the page chain crosses extents and allocation units.

  Buffers brought in by large I/O may be flushed from cache before all of the pages are read. The asynchronous prefetch look-ahead set size is reduced by cross-allocation unit hops while following the page chain.

  For tables that use data-only locking, large I/O and asynchronous prefetch performance suffers because the forward pages cross extents and allocation units.

Once the fragmentation starts to degrade on application performance, perform maintenance, keeping in mind that dropping and recreating clustered indexes requires 120% of the space occupied by the table.

If space is unavailable, maintenance becomes more complex and takes longer. The best, and often cheapest, solution is to add enough disk capacity to provide room for the index creation.

Adding disks when devices are full

Simply adding disks and recreating indexes when partitions are full may not solve load-balancing problems. If a physical device that holds a partition becomes completely full, the data-copy stage of recreating an index cannot copy data to that physical device.

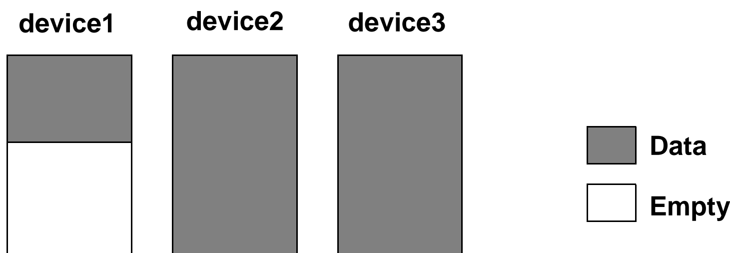
If a physical device is almost completely full, recreating the clustered index does not always succeed in establishing a good load balance.

Adding disks when devices are full

The result of creating a clustered index when a physical device is completely full is that two partitions are created on one of the other physical devices.

devices2 and **device3** are completely full, as shown in Figure 1-2.

Figure 1-2: A table with three partitions on three devices

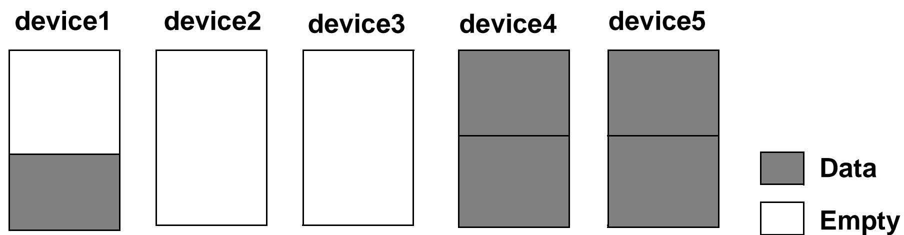


In the example above, adding two devices, repartitioning the table to use five partitions, and dropping and recreating the clustered index produces the following results:

Device 1	One partition, approximately 40% full.
Devices 2 and 3	Empty. These devices had no free space when create index started, so a partition for the copy of the index cannot be created on the device.
Devices 4 and 5	Each device has two partitions, and each is 100% full.

Figure 1-3 shows these results.

Figure 1-3: Devices and partitions after create index

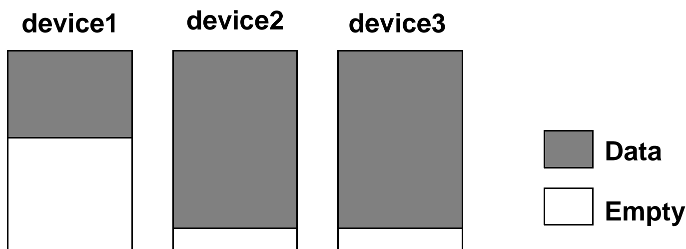


The only solution, once a device becomes completely full, is to bulk-copy the data out, truncate the table, and copy the data into the table again.

Adding disks when devices are nearly full

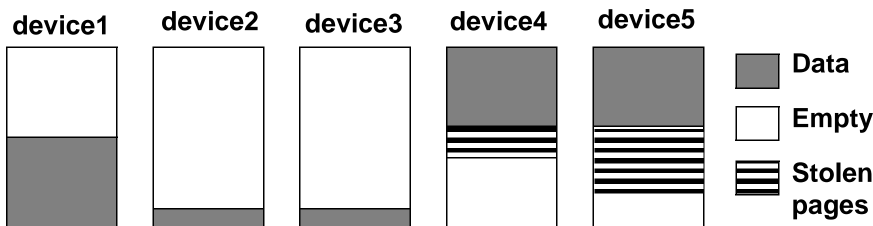
If a device is nearly full, recreating a clustered index does not balance data across devices. Instead, the device that is nearly full stores a small portion of the partition, and the other space allocations for the partition steals extents on other devices. Figure 1-4 shows a table with nearly full data devices.

Figure 1-4: Partitions almost completely fill the devices



After adding devices and recreating the clustered index, the result might be similar to the results shown in Figure 1-5.

Figure 1-5: Extent stealing and unbalanced data distribution



Once the partitions on device2 and device3 use the small amount of space available, they start stealing extents from device4 and device5.

Re-creating the indexes another time may lead to a more balanced distribution. However, if one of the devices is nearly filled by extent stealing, re-creating indexes again does not solve the problem.

Using bulk copy to copy the data out and back in again is most effective solution to this form of imbalance.

To avoid situations such as these, monitor space usage on the devices, and add space early.

Maintenance issues and partitioned tables

Maintenance activity requirements for partitioned tables depend on the frequency and type of updates performed on the table.

Partitioned tables that require little maintenance include:

- Tables that are read-only or that experience very few updates. For tables that have few updates, only periodic checks for balance are required.
- Tables where inserts are well-distributed across the partitions. Random inserts to partitioned heap tables and inserts that are evenly distributed due to a clustered index key that places rows on different partitions do not develop skewed distribution of pages.

  If data modifications lead to space fragmentation and partially filled data pages, you may need to recreate the clustered index.
- Heap tables where inserts are performed by bulk copy. You can use parallel bulk copy to direct the new data to specific partitions to maintain load balancing.

Partitioned tables that require frequent monitoring and maintenance include tables with clustered indexes that tend to direct new rows to a subset of the partitions. An ascending key index is likely to require more frequent maintenance.

Regular maintenance checks for partitioned tables

Routine monitoring for partitioned tables should include the following types of checks, in addition to routine database consistency checks:

- Use sp_helpartition to check the balance on partitions. If some partitions are significantly larger or smaller than the average, recreate the clustered index to redistribute data.
- Use sp_helpsegment to check the balance of space on underlying disks.
- If you recreate the clustered index to redistribute data for parallel query performance, check for devices that are nearing 50% full. Adding space before devices become too full avoids the complicated procedures described earlier in this chapter.
- Use sp_helpsegment to check the space available as free pages on each device, or use sp_helpdb to check for free kilobytes.

You might need to recreate the clustered index on partitioned tables because:

- Your index key tends to assign inserts to a subset of the partitions.
- Delete activity tends to remove data from a subset of the partitions, leading to I/O imbalance and partition-based scan imbalances.
- The table has many inserts, updates, and deletes, leading to many partially filled data pages. This condition leads to wasted space, both on disk and in the cache, and increases I/O because more pages need to be read for many queries.

CHAPTER 2 Data Storage

This chapter explains how Adaptive Server® stores data rows on pages, and how those pages are used in select and data modification statements when there are no indexes. This chapter lays the foundation for understanding how to improve Adaptive Server performance by creating indexes, tuning queries, and addressing object storage issues.

Topic	Page
Query optimization	21
Adaptive Server pages	23
Pages that manage space allocation	26
Space overheads	30
Tables without clustered indexes	38
Caches and object bindings	49

Query optimization

The Adaptive Server optimizer attempts to find the most efficient access path to your data for each table in a query by estimating the cost of the physical I/O needed to access the data, and the number of times each page must be read while in the data cache.

In most database applications, there are many tables in the database, and each table has one or more indexes. Depending on whether you have created indexes, and what kind of indexes you have created, the optimizer's access method options include:

- Table scan – reading all the table's data pages, sometimes hundreds or thousands of pages.
- Index access – using the index to find only the data pages needed, sometimes as few as three or four page reads in all.
- Index covering – using only an index to return data, without reading the actual data rows, requiring only a fraction of the page reads required for a table scan.

Using the appopriate indexes on tables should allow most queries to access the data they need with a minimum number of page reads.

Query processing and page reads

Most of a query's execution time is spent reading data pages from disk. Therefore, most performance improvement comes from reducing the number of disk reads needed for each query.

When a query performs a table scan, Adaptive Server reads every page in the table because no indexes are available to help it retrieve the data. The query may have poor response time, because disk reads take time. Queries that incur costly table scans also affect the performance of other queries on your server.

Table scans can increase the time other users have to wait for a response, since they use system resources such as CPU time, disk I/O, and network capacity.

Table scans use a large number of disk reads (I/Os) for a given query. When you have become familiar with the access methods, tuning tools, the size and structure of your tables, and the queries in your applications, you should be able to estimate the number of I/O operations a given join or select operation will perform, given the indexes that are available.

If you know what the indexed columns on your tables are, and the table and index sizes, you can often look at a query and predict its behavior. For different queries on the same table, you might be able to draw these conclusions:

- This query returns a single row or a small number of rows that match the where clause condition.

  The condition in the where clause is indexed; it should perform two to four I/Os on the index and one more to read the correct data page.

- All columns in the select list and where clause for this query are included in a nonclustered index. This query will probably perform a scan on the leaf level of the index, about 600 pages.

  Adding an unindexed column to the select list would force the query to scan the table, which would require 5000 disk reads.

- No useful indexes are available for this query; it is going to do a table scan, requiring at least 5000 disk reads.

This chapter describes how tables are stored, and how access to data rows takes place when indexes are not being used.

Chapter 5, “Indexes,” in *Performance and Tuning Series: Locking and Concurrency Control* describes access methods for indexes. Chapter 3, “Setting Space Management Properties” and Chapter 4, “Table and Index Size” explain how to determine which access method is being used for a query, the size of the tables and indexes, and the amount of I/O a query performs. These chapters provide a basis for understanding how the optimizer models the cost of accessing the data for your queries.

Adaptive Server pages

These types of pages store database objects:

- Data pages – store the data rows for a table.
- Index pages – store the index rows for all levels of an index.
- Large object (LOB) pages – store the data for text and image columns, and for Java off-row columns.

Adaptive Server versions 12.5 and later do not use the buildmaster binary to build the master device. Instead, Sybase® has incorporated the buildmaster functionality in the dataserver binary.

The dataserver command allows you to create master devices and databases with logical pages of size 2K, 4K, 8K, or 16K. Larger logical pages allow you to create larger rows, which can improve your performance because Adaptive Server accesses more data each time it reads a page. For example, a 16K page can hold 8 times the amount of data as a 2K page, an 8K page holds 4 times as much data as a 2K page, and so on, for all the sizes for logical pages.

The logical page size is a server-wide setting; you cannot have databases with varying size logical pages within the same server. All tables are automatically appropriately sized so that the row size is no greater than the current page size of the server. That is, rows cannot span multiple pages.

See the *Utility Guide* for specific information about using the dataserver command to build your master device.

Adaptive Server may be required to process large volumes of data for a single query, DML operation, or command. For example, if you use a data-only-locked (DOL) 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 means a potential reduction in throughput.

The size of Adaptive Server logical pages determines the server's space allocation. Each allocation page, object allocation map (OAM) page, data page, index page, text page, and so on is built on a logical page. For example, if the logical page size of Adaptive Server is 8K, each of these page types are 8K in size. All of these pages consume the entire size specified by the size of the logical page. OAM pages have a greater number of OAM entries for larger logical pages (for example, 8K) than for smaller pages (2K).

Page headers and page sizes

All pages have a header that stores information, such as the partition ID that the page belongs to, and other information used to manage space on the page. Table 2-1 shows the number of bytes of overhead and usable space on data and index pages for a server configured for 2K pages.

Table 2-1: Overhead and user data space on data and index pages

Locking Scheme	Overhead	Bytes for user data
Allpages	32	2016
Data-only	46	2002

The rest of the page is available to store data and index rows.

For information on how text, image, and Java columns are stored, see "Large object (LOB) pages" on page 25.

Data and index pages

Data pages and index pages on data-only-locked tables have a row offset table that stores pointers to the starting byte for each row on the page. Each pointer takes 2 bytes.

Data and index rows are inserted on a page starting just after the page header, and fill in contiguously. For all tables and indexes on data-only-locked tables, the row offset table begins at the last byte on the page, and grows upward.

The information stored for each row consists of the actual column data, plus information such as the row number and the number of variable-length and null columns in the row. Index pages for allpages-locked tables do not have a row offset table.

Rows cannot cross page boundaries, except for text, image, and Java off-row columns. Each data row has at least 4 bytes of overhead; rows that contain variable-length data have additional overhead.

See Chapter 4, “Table and Index Size,” for more information on data and index row sizes and overhead.

Large object (LOB) pages

text, image, and Java off-row columns for a table are stored as a separate data structure, consisting of a set of pages. These columns are known as large object, or LOB, columns. Each table with a text or image column has one of these structures. If a table has multiple LOB columns, it still has only one of these separate data structures.

The table itself stores a 16-byte pointer to the first page of the value for the row. Additional pages for the value are linked by next and previous pointers. Each value is stored in its own separate page chain. The first page stores the number of bytes in the text value. The last page in the chain for a value is terminated with a null next-page pointer.

Reading or writing a LOB value requires at least two page reads or writes:

- One for the pointer
- One for the actual location of the text in the text object

Each LOB page stores up to 1800 bytes. Every non-null value uses at least one full page.

LOB structures are listed separately in sysindexes. The ID for the LOB structure is the same as the table’s ID. The index ID column is indid and is always 255, and the name is the table name, prefixed with the letter “t”.

Extents

Adaptive Server pages are always allocated to a table, index, or LOB structure. A block of 8 pages is called an **extent**. The size of an extent depends on the page size the server uses. The extent size on a 2K server is 16K; on an 8K it is 64K, and so on. The smallest amount of space that a table or index can occupy is 1 extent, or 8 pages. Extents are deallocated only when all the pages in an extent are empty.

The use of extents in Adaptive Server is transparent to the user except when examining reports on space usage. For example, reports from sp_spaceused display the space allocated (the reserved column) and the space used by data and indexes. The unused column displays the amount of space in extents that are allocated to an object, but not yet used to store data.

```
sp_spaceused titles
name   rowtotal reserved data    index_size unused
------ -------- -------- ------- ---------- ------
titles 5000     1392 KB 1250 KB 94 KB      48 KB
```

In this report, the titles table and its indexes have 1392KB reserved on various extents, including 48KB (24 data pages) that is unallocated.

Note Adaptive Server avoids wasting extra space by filling up existing allocated extents in the target allocation page, even though these extents are assigned to other partitions. The net effect is that extents are allocated only when there are no free extents in the target allocation page

---

Pages that manage space allocation

In addition to data, index, and LOB pages used for data storage, Adaptive Server uses other types of pages to manage storage, track space allocation, and locate database objects. The sysindexes table also stores pointers that are used during data access.

The pages that manage space allocation and the sysindexes pointers are used to:

- Speed the process of finding objects in the database.
- Speed the process of allocating and deallocating space for objects.
- Provide a means for Adaptive Server to allocate additional space for an object that is near the space already used by the object. This helps performance by reducing disk-head travel.

These pages track the disk space use by database objects:

- Global allocation map (GAM) pages contain allocation bitmaps for an entire database.
- Allocation pages track space usage and objects within groups of 256 pages, or .5MB.

- Object allocation map (OAM) pages contain information about the extents used for an object. Each partition of a table and index has at least one OAM page that tracks where pages for the object are stored in the database.
- OAM pages manage space allocation for partitioned tables.

Global allocation map pages

Each database has a GAM, which stores a bitmap for all allocation units of a database, with 1 bit per allocation unit. When an allocation unit has no free extents available to store objects, the corresponding bit in the GAM is set to 1.

This mechanism expedites allocating new space for objects. Users cannot view the GAM page; it appears in the system catalogs as the sysgams table.

Allocation pages

When you create a database or add space to a database, the space is divided into allocation units of 256 data pages. The first page in each **allocation unit** is the allocation page. Page 0 and all pages that are multiples of 256 are allocation pages.

The allocation page tracks space in each extent on the allocation unit by recording the partition ID, object ID, and index ID for the object that is stored on the extent, and the number of used and free pages. The allocation page also stores the page ID for the table or index's corresponding OAM page.

Object allocation map pages

Each partition for a table, index, and text chain has one or more object allocation map (OAM) pages stored on pages allocated to the table or index. If a table has more than one OAM page, the pages are linked in a chain. OAM pages store pointers to the allocation units that contain pages for the object.

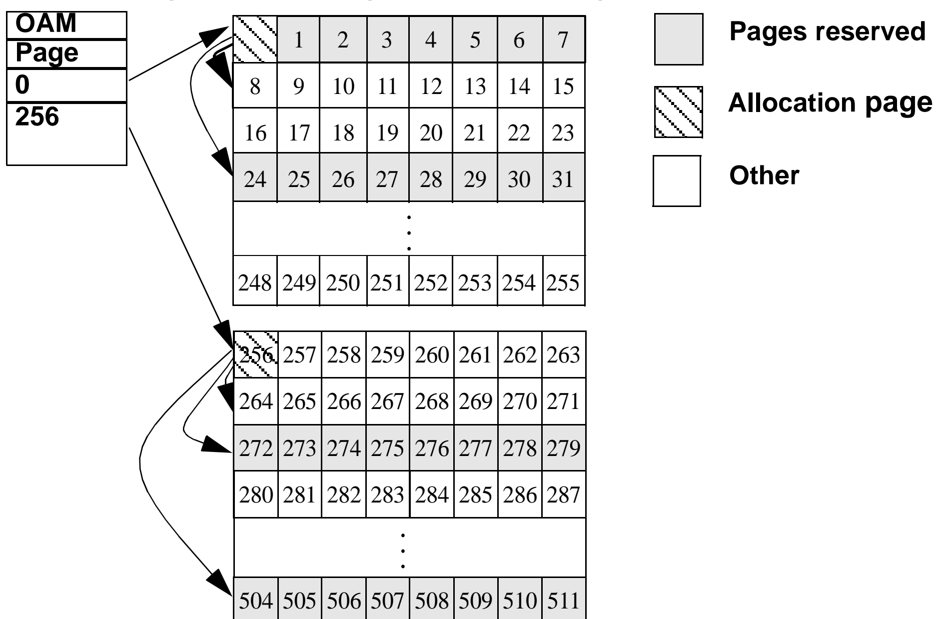
The first page in the chain stores allocation hints, indicating which OAM page in the chain stores information about allocation units with free space. This provides a fast way to allocate additional space for an object and to keep the new space close to pages already used by the object.

How OAM pages and allocation pages manage object storage

Figure 2-1 shows how allocation units, extents, and objects are managed by OAM pages and allocation pages.

- Two allocation units are shown, one starting at page 0 and one at page 256. The first page of each is the allocation page.
- A table is stored on four extents, starting at pages 1 and 24 on the first allocation unit and pages 272 and 504 on the second unit.
- The first page of the table is the table's OAM page. It points to the allocation page for each allocation unit where the object uses pages, so it points to pages 0 and 256.
- Allocation pages 0 and 256 store the table's object ID, index ID, and partition ID to which the extent belongs. Allocation page 0 points to pages 1 and 24 for the table, and allocation page 256 points to pages 272 and 504.

Figure 2-1: OAM page and allocation page pointers



Page allocation keeps an object's pages together

Adaptive Server attempts to keep page allocations close together for each object (for example, a table partition, index, or a table's text or image chain).

Typically, when Adaptive Server requires a new page:

- If the object's current extent contains a free page, Adaptive Server uses this page.
- If the current extent has no free pages, but another extent assigned to the object on the same allocation unit has a free page, Adaptive Server uses this page.
- If the current allocation unit has no extents with free pages assigned to the object, but has a free extent, Adaptive Sever allocates the first available page of the free extent.
- If the current allocation unit is full, Adaptive Server scans the object's OAM page for another allocation unit containing extents with free pages, and uses the first available page.
- If no OAM entries indicate available free pages, Adaptive Server compares the OAM entries to the global allocation map page to see if any allocation units have free extents. Adaptive Server allocates the first available page of first free extent.
- If all of the OAM entries are for full allocation units, Adaptive Server searches the global allocation map for an allocation unit with at least one free extent. Adaptive Server adds a new OAM entry for that allocation unit to the object's OAM, allocates a free extent, and uses the first free page on the extent.

Note Operations like bcp and reorg rebuild using large scale allocation do not look for free pages on already allocated extents; instead they allocate full free extents. Large scale allocation cannot typically use the first extent of each allocation unit. The first extent only has 7 usable pages because its first page holds the allocation page structure.

Data access using *sysindexes* and *syspartitions*

The sysindexes table stores information about indexed and nonindexed tables. sysindexes has one row for each:

- Allpages-locked table – the indid column is 0 if the table does not have a clustered index, and 1 if the table does have a clustered index.
- Data-only-locked tables – the indid is always 0 for the table.
- Nonclustered index – and for each clustered index on a data-only-locked table.
- Table with one or more LOB columns – the index ID is always 255 for the LOB structure.

syspartitions stores information about each table and index partition and includes one row per partition.

In Adaptive Server version 15.0. and later, each row in syspartitions stores pointers to a table or index to speed access to objects. Table 2-2 shows how these pointers are used during data access.

Table 2-2: Use of syspartitions pointers in data access

Column	Use for table access	Use for index access
root	If indid is 0 and the table is a partitioned allpages-locked table, root points to the last page of the heap.	Used to find the root page of the index tree.
first	Points to the first data page in the page chain for allpages-locked tables.	Points to the first leaf-level page in a nonclustered index, or a clustered index on a data-only-locked table.
doampg	Points to the first OAM page for the table.	
ioampg		Points to the first OAM page for an index.

Space overheads

Regardless of the logical page size for which it is configured, Adaptive Server allocates space for objects (tables, indexes, text page chains) in extents, each of which is eight logical pages. That is, if a server is configured for 2K logical pages, it allocates one extent, 16K, for each of these objects; if a server is configured for 16K logical pages, it allocates one extent, 128K, for each of these objects.

This is also true for system tables. If your server has many small tables, space consumption can be quite large if the server uses larger logical pages.

For example, for a server configured for 2KB logical pages, systypes—with approximately 31 short rows, a clustered and a non-clustered index—reserves 3 extents, or 48KB of memory. If you migrate the server to use 8KB pages, the space reserved for systypes is still 3 extents, 192KB of memory.

For a server configured for 16KB, systypes requires 384KB of disk space. For small tables, the space unused in the last extent can become significant on servers using larger logical page sizes.

Databases are also affected by larger page sizes. Each database includes the system catalogs and their indexes. If you migrate from a smaller to larger logical page size, you must account for the amount of disk space each database requires.

Number of columns and size

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

- 1024 for fixed-length columns in both allpages-locked (APL) and data-only-locked (DOL) tables
- 254 for variable-length columns in an APL table
- 1024 for variable-length columns in an DOL table

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 APL 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 a APL table can be as large as 4010 bytes, less the row format overheads. See Table 2-3 for more information.
- If you attempt to create a table with a fixed-length column that is greater than the limits of the logical page size, create table issues an error message.

Table 2-3: Maximum row and column length

Locking scheme	Page size	Maximum row length	Maximum column length
	2K (2048 bytes)	1962	1960 bytes
	4K (4096 bytes)	4010	4008 bytes
APL tables	8K (8192 bytes)	8106	8104 bytes
	16K (16384 bytes)	16298	16296 bytes
	2K (2048 bytes)	1964	1958 bytes
	4K (4096 bytes)	4012	4006 bytes
DOL tables	8K (8192 bytes)	8108	8102 bytes
	16K (16384 bytes)	16300	16294 bytes if table does not include any variable-length columns
	16K (16384 bytes)	16300 (subject to a *max start* offset of varlen = 8191)	8191-6-2 = 8183 bytes if table includes at least on variable-length column.

The maximum size of a fixed-length column in a DOL table with a 16K logical page size depends on whether the table contains variable-length columns. The maximum possible starting offset of a variable-length column is 8191. If the table has any variable-length columns, the sum of the fixed-length portion of the row, plus overheads, cannot exceed 8191 bytes, and the maximum possible size of all the fixed-length columns is restricted to 8183 bytes, when the table contains any variable-length columns.

In Table 2-3, the maximum column length is determined by subtracting 6 bytes for row overhead and 2 bytes for row length field.

Variable-length columns in APL tables

APL tables that contain one variable-length column (for example, varchar, varbinary and so on) have the following minimum overhead for each row:

- 2 bytes for the initial row overhead.
- 2 bytes for the row length.
- 2 bytes for the column-offset table at the end of the row. This is always *n+1* bytes, where *n* is the number of variable-length columns in the row.

A single-column table has an overhead of at least 6 bytes, plus additional overhead. The maximum column size, after overhead, must be less than or equal to:
(column length) + (additional overhead) + (six-byte overhead.)

Table 2-4: Maximum size for variable-length columns in an APL table

Page size	Maximum row length	Maximum column length
2K (2048 bytes)	1962	1948
4K (4096 bytes)	4010	3988
8K (8192 bytes)	8096	8068
16K (16384 bytes)	16298	16228

Variable-length columns that exceed the logical page size

If your table uses 2K logical pages, you can create some variable-length columns whose total row-length exceeds the maximum row-length for a 2K page size. This allows you to create tables where some, but not all, variable-length columns contain the maximum possible size. However, when you issue create table, you receive a warning message that says the resulting row size may exceed the maximum possible row size, and cause a future insert or update to fail.

For example, if you create a table that uses a 2K page size, and contains a variable-length column with a length of 1975 bytes, Adaptive Server creates the table but issues a warning message. You cannot insert data that exceeds the maximum length of the row (1962 bytes).

Variable-length columns in DOL tables

For a single, variable-length column in a DOL table, the minimum overhead for each row is:

- Six bytes for the initial row overhead.
- Two bytes for the row length.
- Two bytes for the column offset table at the end of the row. Each column offset entry is two bytes. There are *n* such entries, where n is the number of variable-length columns in the row.

The total overhead is 10 bytes. There is no adjust table for DOL rows. The actual variable-length column size is:

```
column length + 10 bytes overhead
```

Table 2-5: Maximum size for variable-length columns in an DOL table

Page size	Maximum row length	Maximum column length
2K (2048 bytes)	1964	1954
4K (4096 bytes)	4012	4002
8K (8192 bytes)	8108	8098
16K (16384 bytes)	16300	16290

DOL tables with variable-length columns must have an offset of fewer than 8191 bytes for all inserts to succeed. For example, this insert fails because the offset for columns c2, c3, and c4 is 9010, which exceeds the maximum of 8191 bytes:

```
create table t1(
    c1 int not null,
    c2 varchar(5000) not null
    c3 varchar(4000) not null
    c4 varchar(10) not null
    ... more fixed length columns)
cvarlen varchar(nnn)) lock datarows
```

Wide, variable-length rows

Adaptive Server allows data-only locked (DOL) columns to use a row offset of up to 32767 bytes for wide, variable-length DOL rows if it is configured for a logical page size of 16K.

Enable wide, variable-length DOL rows for each database using:

```
sp_dboption database_name, 'allow wide dol rows', true
```

Note allow wide dol rows is on by default for temporary databases. You cannot set allow wide dol rows for the master database.

---

sp_dboption 'allow wide dol rows' has no effect on user databases with logical page sizes smaller than 16K; Adaptive Server cannot create a wide, variable-length DOL rows on pages smaller than 16384 bytes.

This example configures the pubs2 database on a server using a page size of 16K to use wide, variable-length DOL rows:

1 Enable wide, variable-length rows for the pubs2 database:

```
sp_dboption pubs2, 'allow wide dol rows', true
```

2 Create the book_desc table, which includes wide, variable-length DOL columns that begin after the row offset of 8192:

```
create table book_desc
(title varchar(80) not null,
title_id varchar(6) not null,
author_desc char(8192) not null,
book_desc varchar(5000) not null)
lock datarows
```

Bulk-copying wide data

You must use the version of bcp shipped with Adaptive Server version 15.7 and later to bulk-copy-in data that contains wide, variable-length DOL rows. You must configure the database receiving the data to accept wide, variable-length DOL rows (that is, bcp does not copy wide rows into databases for which you have not enabled allow wide dol rows).

Checking for downgrade

See the Installation Guide for your platform for details about downgrading an Adaptive Server that uses wide, variable-length rows.

Dumping and loading wide, variable-length DOL columns

Database and transaction log dumps retain their setting for allow wide dol rows, which is imported into the database into which you are loading the dump (if this database does not already have the option set).

For example, if you load a transaction log dump named my_table_log.dmp, which has allow wide dol rows set to true, into database big_database, for which you have not set allow wide dol rows, my_table.log retains its setting of true for allow wide dol rows after the load to big_database completes.

However, if the database or transaction log dump does not have allow wide dol rows set, but the database into which you are loading the dump does, allow wide dol rows remains set.

You cannot load a dump of a database that has allow wide dol rows enabled to versions of Adaptive Server earlier than 15.7.

Using proxy tables with wide, variable-length DOL rows

You can use proxy tables with wide, variable-length DOL rows.

When you create proxy tables (regardless of their row length), the controlling Adaptive Server is the one on which you execute the create table or create proxy table command. Adaptive Server executes these commands on the server to which you are connected. However, Adaptive Server executes data manipulation statements (insert, delete) on the server on which the data is stored, and the user's local server only formats the request and then sends it; the local server does not control anything.

Adaptive Server creates proxy tables as though they are created on the local server, even though the data resides on remote servers. If the create proxy_table command creates a DOL table that contains wide, variable-length rows, the command succeeds only if the database in which you are creating the proxy table has allow wide dol rows set to true.

Note Adaptive Server creates proxy tables using the local server's lock scheme configuration, and creates a proxy table with DOL rows if lock scheme is set to datarows or datapages.

When you insert or update data in proxy tables, Adaptive Server ignores the local database's setting for allow wide dol rows. The server where the data resides determines whether an insert or update succeeds

Restrictions for converting locking schemes or using select into

The following restrictions apply whether you are using alter table to change a locking scheme or using select into to copy data into a new table.

For servers that use page sizes other than 16K pages, the maximum length of a variable-length column in an APL table is less than that for a DOL table, so you can convert the locking scheme of an APL table with a maximum sized variable-length column to DOL. You cannot, however, convert a DOL table containing at least one maximum sized variable-length column to an APL table.

On servers that use 16K pages, APL tables can store substantially larger sized variable-length columns than DOL tables. You can convert tables from DOL to APL, but you cannot convert from APL to DOL.

These restrictions on locking-scheme conversions occur only if data in the source table exceeds the limits of the target table. If this occurs, Adaptive Server raises an error message while transforming the row format from one locking scheme to the other. If the table is empty, no such data transformation is required, and the lock-change operation succeeds. However, subsequent inserts or updates to the table, users may see errors caused by limitations on the column or row size for the target schema of the altered table.

Organizing columns in DOL tables by size of variable-length columns

For DOL tables that use variable-length columns, arrange the columns so that the longest columns are placed toward the end of the table definition. This allows you to create tables with much larger rows than if the large columns appear at the beginning of the table definition. For instance, in a 16K page server, the following table definition is acceptable:

```
create table t1 (
    c1 int not null,
    c2 varchar(1000) null,
    c3 varchar(4000) null,
    c4 varchar(9000) null) lock datarows
```

However, the following table definition typically is unacceptable for future inserts. The potential start offset for column c2 is greater than the 8192-byte limit because of the preceding 9000-byte c4 column:

```
create table t2 (
    c1 int not null,
    c4 varchar(9000) null,
    c3 varchar(4000) null,
    c2 varchar(1000) null) lock datarows
```

The table is created, but future inserts may fail.

Number of rows per data page

The number of rows allowed for a DOL data page is determined by:

- The page size
- A 10-byte overhead for the row ID, which specifies a row-forwarding address

Table 2-6 displays the maximum number of rows that can fit on a DOL data page:

Table 2-6: Maximum number of data rows for a DOL data page

Page size	Maximum number of rows
2K	166
4K	337
8K	678
16K	1361

APL data pages can have a maximum of 256 rows. Because each page requires a one-byte row number specifier, large pages with short rows incur some unused space. For example, if Adaptive Server is configured with 8K logical pages and rows that are 25 bytes, each page has 1275 bytes of unused space, after accounting for the row-offset table and the page header.

Additional number of object and size restrictions

The maximum number of arguments for stored procedures is 2048. See Chapter 16, "Using Stored Procedures," in the *Transact-SQL Users Guide*.

Adaptive Server version 12.5 and later can store data that has different limits than data stored in versions earlier than 12.5. Clients must be able to store and process these newer data limits. If you are using older versions of Open Client and Open Server, they cannot process the data if you:

- Upgrade to Adaptive Server version 12.5 or later
- Drop and recreate the tables with wide columns
- Insert wide data

See Chapter 2, "Basic Configuration for Open Client" in the *Open Client Configuration Guide*.

Tables without clustered indexes

If you create a table on Adaptive Server, but do not create a clustered index, the table is stored as a heap, which means the data rows are not stored in any particular order. This section describes how select, insert, delete, and update operations perform on heap tables when there is no "useful" index to aid in retrieving data.

There are very few justifications for heap tables. Most applications perform better with clustered indexes on the tables. However, heap tables work well for small tables that use only a few pages, and for tables where changes are infrequent

Heap tables can be useful for tables that do not require:

- Direct access to single, random rows
- Ordering of result sets

Heap tables do not work well for queries against most large tables that must return a subset of the table's rows.

Partitioned heap tables are useful in applications with frequent, large volumes of batch inserts where the overhead of dropping and creating clustered indexes is unacceptable.

Sequential disk access is efficient, especially with large I/O and asynchronous prefetch. However, the entire table must always be scanned to find any value, and this has potentially large impact in the data cache and other queries.

Batch inserts can also perform efficient sequential I/O. However, there is a potential bottleneck on the last page if multiple processes try to insert data concurrently.

Sometimes, an index exists on the columns named in a where clause, but the optimizer determines that it would be more costly to use the index than to perform a table scan.

Table scans are always used when you select all rows in a table. The only exception is when the query includes only columns that are keys in a nonclustered index.

For more information, see Chapter 5, "Indexes," in *Performance and Tuning Series: Locking and Concurrency Control*.

Locking schemes

The data pages in an APL table are linked into a list of pages by pointers on each page. Pages in data-only-locked tables are not linked into a page chain.

In an allpages-locked table, each page stores a pointer to the next page in the chain and to the previous page in the chain. When you insert new pages, the pointers on the two adjacent pages change to point to the new page. When Adaptive Server scans an allpages-locked table, it reads the pages in order, following these page pointers.

Pages are also doubly linked at each index level of allpages-locked tables, and at the leaf level of indexes on data-only-locked tables. If an allpages-locked table is partitioned, there is one page chain for each partition.

Unlike allpages-locked tables, data-only-locked tables typically do not maintain a page chain, except immediately after you create a clustered index. However, this page chain is broken the first time you issue a command on the table.

When Adaptive Server scans a data-only-locked table, it uses the information stored in the OAM pages. See "Object allocation map pages" on page 27.

Another difference between allpages-locked tables and data-only-locked tables is that data-only-locked tables use fixed row IDs. This means that row IDs (a combination of the page number and the row number on the page) do not change in a data-only-locked table during normal query processing.

Row IDs change only when one of the operations that require data-row copying is performed, for example, during reorg rebuild or while creating a clustered index.

For information on how fixed row IDs affect heap operations, see "Deleting from a data-only locked heap table" on page 43 and "Data-only-locked heap tables" on page 44.

Select operations on heap tables

When you issue a select query on a heap, and there is no useful index, Adaptive Server must scan every data page in the table to find every row that satisfies the conditions in the query. There may be one row, many rows, or no rows that match.

Allpages-locked heap tables

For allpages-locked tables, Adaptive Server reads the firstpage column in syspartitions for the table, reads the first page into cache, and follows the next page pointers until it finds the last page of the table.

Data-only locked heap tables

Since the pages of data-only-locked tables are not linked in a page chain, a select query on a data-only-locked heap table uses the table's OAM and the allocation pages to locate all the rows in the table. The OAM page points to the allocation pages, which point to the extents and pages for the table.

Inserting data into an allpages-locked heap table

When you insert data into an allpages-locked heap table without a clustered index, the data row is always added to the last page of the table. If there is no clustered index on a table, and the table is not partitioned, the syspartitions.root entry for the heap table stores a pointer to the last page of the heap to indicate the page where the data should be inserted.

If the last page is full, a new page is allocated in the current extent and linked onto the chain. If the extent is full, Adaptive Server looks for empty pages on other extents being used by the table. If no pages are available, a new extent is allocated to the table.

One of the severe performance limits on heap tables that use allpages locking is that the page must be locked when the row is added, and the lock is held until the transaction completes. If many users are trying to insert into an allpages-locked heap table simultaneously, each insert must wait for the preceding transaction to complete.

This problem of last-page conflicts on heap tables is true for:

- Single-row inserts
- Multiple row inserts using select into or insert...select, or several insert statements in a batch
- Bulk copying into the table

To address last-page conflicts on heap tables, try:

- Switching to datapages or datarows locking
- Creating a clustered index that directs the inserts to different pages
- Partitioning the table, which creates multiple insert points for the table, giving you multiple "last pages" in an allpages-locked table

For all transactions where there may be lock conflicts, you can also:

- Keep transactions short

- Avoid network activity and user interaction whenever possible, once a transaction acquires locks

Inserting data into a data-only-locked heap table

When users insert data into a data-only-locked heap table, Adaptive Server tracks page numbers where the inserts have recently occurred, and keeps the page number as a suggestion for future tasks that need space. Subsequent inserts to the table are directed to one of these pages. If the page is full, Adaptive Server allocates a new page and replaces the old hint with the new page number.

Blocking while many users are simultaneously inserting data is much less likely to occur during inserts to data-only-locked heap tables than in APL tables. When blocking does occur, Adaptive Server allocates a small number of empty pages and directs new inserts to those pages using these newly allocated pages as hints.

For datarows-locked tables, blocking occurs only while the actual changes to the data page are being written; although row locks are held for the duration of the transaction, other rows can be inserted on the page. The row-level locks allow multiple transaction to hold locks on the page.

There may be slight blocking on data-only-locked tables, because Adaptive Server allows a small amount of blocking after many pages have just been allocated, so that the newly allocated pages are filled before additional pages are allocated.

Conflicts during inserts to heap tables are greatly reduced for data-only-locked tables, but can still take place. If these conflicts slow inserts, try:

- Switching to datarows locking, if the table uses datapages locking
- Using a clustered index to spread data inserts
- Partitioning the table, which provides additional hints and allows new pages to be allocated on each partition when blocking takes place

Deleting data from a heap table

When you delete rows from a heap table that does not have a useful index, Adaptive Server scans the data rows in the table to find the rows to delete. It cannot determine how many rows match the conditions in the query without examining every row.

Deleting from an allpages-locked heap table

When a data row is deleted from a page in an allpages-locked table, the rows that follow it on the page move up so that the data on the page remains contiguous, avoiding fragmentation within the page.

Deleting from a data-only locked heap table

When you delete rows from a data-only-locked heap table, a table scan is required if there is no useful index. The OAM and allocation pages are used to locate the pages.

The space on the page is not recovered immediately. Rows in data-only-locked tables must maintain fixed row IDs, and must be reinserted in the same place if the transaction is rolled back.

After a delete transaction commits, one of the following processes shifts rows on the page to make the space usage contiguous:

- The housekeeper garbage collection process
- An insert that needs to find space on the page
- The reorg reclaim_space command

Deleting the last row on a page

If you delete the last row on a page, the page is deallocated. If other pages on the extent are still in use by the table, the page can be used again by the table when a page is needed.

If all other pages on the extent are empty, the entire extent is deallocated. It can be allocated to other objects in the database. The first data page for a table or an index is never deallocated.

Updating data on a heap table

Like other operations on heap tables, an update on a table that has no useful index on the columns in the where clause performs a table scan to locate the rows to be changed.

Allpages-locked heap tables

You can perform updates on allpages-locked heap tables in several ways:

- If the length of the row does not change, the updated row replaces the existing row, and no data moves on the page.
- If the length of the row changes, and there is enough free space on the page, the row remains in the same place on the page, but other rows move up or down to keep the rows contiguous on the page.

  The row offset pointers at the end of the page are adjusted to point to the changed row locations.
- If the row does not fit on the page, the row is deleted from its current page, and inserted as a new row on the last page of the table. This type of update may cause a conflict on the last page of the heap.

Data-only-locked heap tables

One of the requirements for data-only-locked tables is that the row ID of a data row never changes (except during intentional rebuilds of the table). Therefore, updates to data-only-locked tables can be performed by the first two methods described above, as long as the row fits on the page.

However, when a row in a data-only-locked table is updated so that it no longer fits on the page, a process called **row forwarding** performs these steps:

1. The row is inserted onto a different page, and
2. A pointer to the row ID on the new page is stored in the original location for the row.

Indexes need not be modified when rows are forwarded. All indexes still point to the original row ID.

If a row must be forwarded a second time, the original location is updated to point to the new page—the forwarded row is never more than one hop away from its original location.

Row forwarding increases concurrency during update operations because indexes do not have to be updated. It can slow data retrieval, however, because a task must read the page at the original location and then read the page where the forwarded data is stored.

Use the reorg command to clear forwarded rows from a table.

See Chapter 1, "Understanding Query Processing" in *Performance and Tuning Series: Query Processing and Abstract Plans*.

How Adaptive Server performs I/O for heap operations

When a query needs a data page, Adaptive Server first checks to see if the page is available in a data cache. If the page is not available, then it must be read from disk. A newly installed Adaptive Server with a 2K logical page size has a single data cache configured for 2K I/O. Each I/O operation reads or writes a single Adaptive Server data page. A system administrator can:

- Configure multiple caches
- Bind tables, indexes, or text chains to the caches
- Configure data caches to perform I/O in page-sized multiples, up to eight data pages (one extent)

To use these caches most efficiently, and to reduce I/O operations, the Adaptive Server optimizer can:

- Choose to prefetch up to eight data pages at a time
- Choose between different caching strategies

Sequential prefetch, or large I/O

Adaptive Server data caches can be configured to allow large I/Os. When a cache allows large I/Os, Adaptive Server can prefetch data pages.

Caches have buffer pools that depend on the logical page sizes, allowing Adaptive Server to read up to an entire extent (eight data pages) in a single I/O operation.

Since much of the time required to perform I/O operations is taken up in seeking and positioning, reading eight pages in a 16K I/O takes nearly the same amount of time as a single page, 2K I/O. Reading eight pages using eight 2K I/Os is nearly eight times more costly than reading eight pages using a single 16K I/O. Table scans perform much better when you use large I/Os.

When several pages are read into cache with a single I/O, they are treated as a unit: they age in cache together, and if any page in the unit has been changed while the buffer was in cache, all pages are written to disk as a unit.

See Chapter 5, "Memory Use and Performance," in *Performance and Tuning Series: Basics*.

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.

Maintaining heap tables

Over time, I/O on heap tables can become inefficient as storage becomes fragmented. Deletes and updates can result in:

- Many partially filled pages
- Inefficient large I/O, since extents may contain many empty pages
- Forwarded rows in data-only-locked tables

To reclaim space in heap tables:

- Use the reorg rebuild command (data-only-locked tables only)
- Create and then drop a clustered index
- Use bcp (the bulk copy utility) and truncate table

Using *reorg rebuild* to reclaim space

reorg rebuild copies all data rows to new pages and rebuilds any indexes on the heap table. You can use reorg rebuild only on data-only-locked tables.

Reclaiming space by creating a clustered index

To create a clustered index, you must have free space in the database that is at least 120% of the table size.

See "Determining the space available for maintenance activities" on page 118.

Reclaiming space using *bcp*

To reclaim space with bcp:

1. Use bcp to copy the table to a file.
2. Use truncate table to truncate the table, reclaiming unused space.
3. Copy the table back in again with bcp.

For detailed information about working with partitioned tables, see Chapter 10, "Partitioning Tables and Indexes," in the *Transact-SQL Users Guide*.

For more information on bcp, see the *Utility Guide*.

Transaction log: a special heap table

The Adaptive Server transaction log is a special heap table that stores information about data modifications in the database. The transaction log is always a heap table; each new transaction record is appended to the end of the log. The transaction log has no indexes.

Place logs on separate physical devices from the data and index pages. Since the log is sequential, the disk head on the log device rarely needs to perform seeks, and you can maintain a high I/O rate to the log.

Transaction log writes occur frequently. Do not let them contend with other I/O in the database, which usually happens at scattered locations on the data pages.

Besides recovery, these operations read the transaction log:

- Any data modification performed in deferred mode.
- Triggers that contain references to the inserted and deleted tables. These tables are built from transaction log records when the tables are queried.
- Transaction rollbacks.

In most cases, the transaction log pages for these queries are still available in the data cache when Adaptive Server needs to read them, and disk I/O is not required.

See "Keeping transaction logs on a separate disk" on page 6 for information about how to improve the performance of the transaction log.

Asynchronous prefetch and I/O on heap tables

Any task that must perform a physical I/O relinquishes the server's engine (CPU) while it waits for the I/O to complete. If a table scan must read 1000 pages, and none of those pages are in cache, performing 2K I/O with no asynchronous prefetch means the task makes 1000 loops, executing on the engine, and then sleeping to wait for I/O. Using 16K I/O requires only 125 loops

Asynchronous prefetch speed the performance of queries that perform table scans. Asynchronous prefetch can request all of the pages on an allocation unit that belong to a table when the task fetches the first page from the allocation unit. If the 1000-page table resides on just 4 allocation units, the task requires many fewer cycles through the execution and sleep loops.

Type of I/O	Loops	Steps in each loop
2K I/O No prefetch	1000	1 Request a page. 2 Sleep until the page has been read from disk. 3 Request a page. 4 Wait for a turn to run on the Adaptive Server engine (CPU). 5 Read the rows on the page.
16K I/O No prefetch	125	1 Request an extent. 2 Sleep until the extent has been read from disk. 3 Wait for a turn to run on the Adaptive Server engine (CPU). 4 Read the rows on the eight pages.
Prefetch	4	1 Request all the pages in an allocation unit. 2 Sleep until the first page has been read from disk. 3 Wait for a turn to run on the Adaptive Server engine (CPU). 4 Read all the rows on all the pages in cache.

Actual performance depends on cache size and other activity in the data cache.

See Chapter 6, "Tuning Asynchronous Prefetch," in *Performance and Tuning Series: Basics*.

Caches and object bindings

A table can be bound to a specific cache. If a table is not bound to a specific cache, but its database is bound to a cache, all of its I/O takes place in that cache.

Otherwise, the table's I/O takes place in the default data cache. You can configure the default data cache for large I/O. Applications that use heap tables are likely to give the best performance when they use a cache configured for 16K I/O.

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

Heap tables, I/O, and cache strategies

Each Adaptive Server data cache is managed as an MRU/LRU (most recently used/least recently used) chain of buffers. As buffers age in the cache, they move from the MRU end toward the LRU end.

When changed pages in the cache pass a point called the **wash marker**, on the MRU/LRU chain, Adaptive Server initiates an asynchronous write on any pages that have changed while they were in cache. This helps ensure that when the pages reach the LRU end of the cache, they are clean and can be reused.

Adaptive Server has two major strategies for using its data cache efficiently:

- LRU replacement strategy
- MRU, or fetch-and-discard replacement strategy

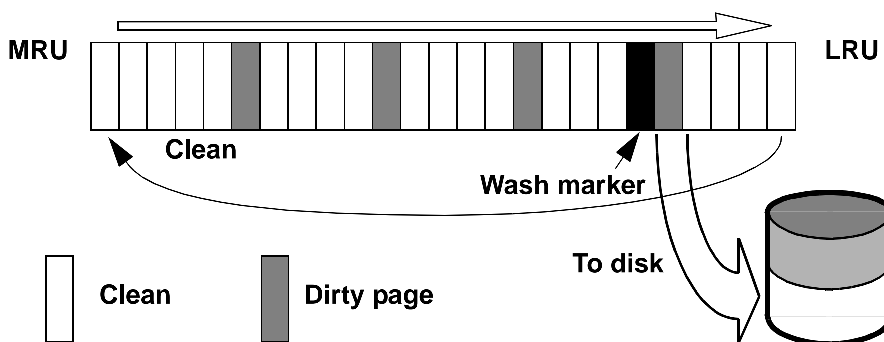
LRU replacement strategy

Adaptive Server uses LRU strategy for:

- Statements that modify data on pages
- Pages that are needed more than once by a single query
- OAM pages
- Most index pages
- Any query where LRU strategy is specified

LRU replacement strategy reads the data pages sequentially into the cache, replacing a “least recently used” buffer. The buffer is placed on the MRU end of the data buffer chain. It moves toward the LRU end as more pages are read into the cache.

Figure 2-2: LRU strategy takes a clean page from the LRU end of the cache



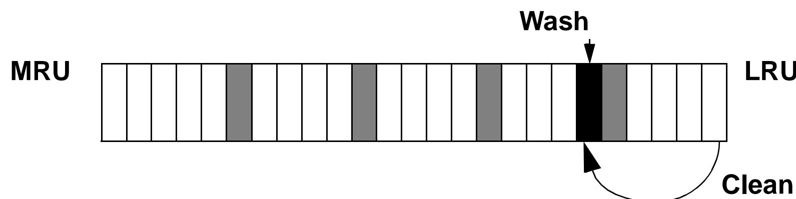
MRU replacement strategy

MRU (fetch-and-discard) is most often used for queries where a page is needed only once by the query, including:

- Most table scans in queries that do not use joins
- One or more tables in a join query

MRU replacement strategy is used for table scanning on heap tables. This strategy places pages into the cache just before the wash marker, as shown in Figure 2-3.

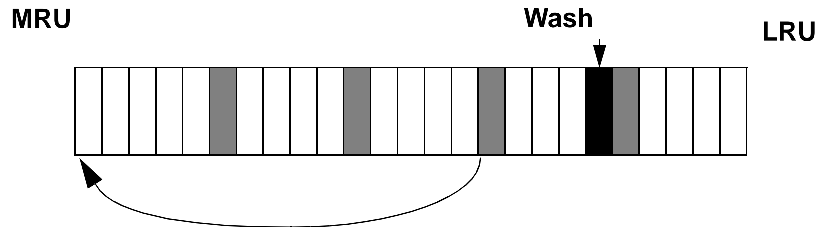
Figure 2-3: MRU strategy places pages just before the wash marker



Placing the pages needed only once at the wash marker means they do not push other pages out of the cache.

The fetch-and-discard strategy is used only on pages actually read from the disk for the query. If a page is already in cache due to earlier activity on the table, the page is placed at the MRU end of the cache.

Figure 2-4: Finding a needed page in cache



Select operations and caching

Under most conditions, single-table select operations on a heap use:

- The largest I/O available to the table, and
- Fetch-and-discard (MRU) replacement strategy

For heap tables, select operations performing large I/O can be very effective. Adaptive Server can read sequentially through all the extents in a table.

See Chapter 1, "Understanding Query Processing" in *Performance and Tuning Series: Query Processing and Abstract Plans*.

Unless the heap is being scanned as the inner table of a nested-loop join, data pages are needed only once for the query, so MRU replacement strategy reads and discards the pages from cache.

Note Large I/O on allpages-locked heap tables is effective only when the page chains are not fragmented. See "Maintaining heap tables" on page 46.

Data modification and caching

Adaptive Server tries to minimize disk writes by keeping changed pages in cache. Many users can make changes to a data page while it resides in the cache. The changes are logged in the transaction log, but the changed data and index pages are not immediately written to disk.

Caching and inserts on heap tables

Inserts to heap tables take place:

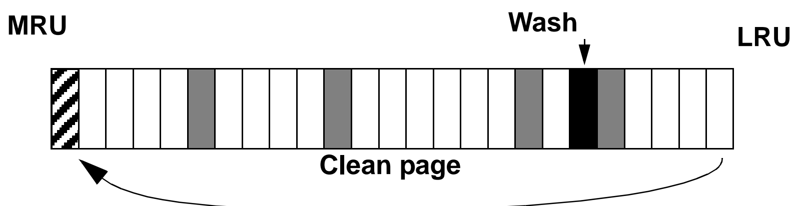
- On the last page of a table that uses allpages locking
- On a page that was recently used for a successful insert, on a table that uses data-only-locking

If an insert is the first row on a new page for the table, a clean data buffer is allocated to store the data page, as shown in Figure 2-5. This page starts to move down the MRU/LRU chain in the data cache as other processes read pages into memory.

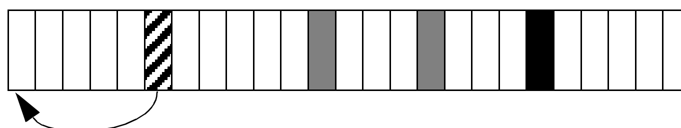
If a second insert to the page takes place while the page is still in memory, the page is located in cache, and moves back to the top of the MRU/LRU chain.

Figure 2-5: Inserts to a heap page in the data cache

First insert on a page takes a clean page from the LRU and puts it on the MRU



Second insert on a page finds the page in cache, and puts in back at the MRU



The changed data page remains in cache until it reaches the LRU end of the chain of pages. The page may be changed or referenced many times while it is in the cache, but it is written to disk only when one of the following takes place:

- The page moves past the wash marker.
- A checkpoint or the housekeeper wash task writes it to disk.

See Chapter 5, “Memory Use and Performance” in *Performance and Tuning Series: Basics*.

Caching, update, and delete operations on heap tables

When you update or delete a row from a heap table, the effects on the data cache are similar to the process for inserts. If a page is already in the cache, the row is changed and the entire buffer (a single page or more, depending on the I/O size) is placed on the MRU end of the chain.

If the page is not in cache, it is read from disk into cache and examined to determine whether the rows on the page match query clauses. Its placement on the MRU/LRU chain depends on whether data on the page needs to be changed:

- If data on the page needs to be changed, the buffer is placed on the MRU end. It remains in cache, where it can be updated repeatedly, or read by other users before being flushed to disk.
- If data on the page does not need to be changed, the buffer is placed immediately before the wash marker in the cache.

CHAPTER 3 Setting Space Management Properties

Setting space management properties can help reduce the amount of maintenance work required to maintain high performance for tables and indexes

This chapter describes the major space management properties for controlling space usage, how these properties affect space usage, and how you can apply them to different tables and indexes.

Topic	Page
Reducing index maintenance	55
Reducing row forwarding	62
Leaving space for forwarded rows and inserts	68
Using max_rows_per_page on allpages-locked tables	75

Reducing index maintenance

The fillfactor option for the create index command allows you to specify how full to make index pages and the data pages of clustered indexes. When you specify a fillfactor value of any amount other than 100%, data and index rows use more disk space than the default setting requires.

If you are creating indexes for tables that will grow in size, you can reduce the impact of page splitting on your tables and indexes by using the fillfactor option.

fillfactor is used when you create an index, and again when you use reorg rebuild to rebuild indexes as part of table reorganization operations (for example, when you rebuild clustered indexes or run reorg rebuild on a table). fillfactor values are not saved in sysindexes, and the fullness of the data or index pages is not maintained over time. fillfactor is not maintained over time during subsequent inserts or updates to the table.

If the leaf-level pages of your index are initially only partially full (because of the fillfactor value), but this free space is used because of subsequent insertions, the leaf-level pages are prone to future splits. Use reorg rebuild...index to build the leaf-level pages, creating them with the specified value for fillfactor so that future insertions do not cause these splits. Run reorg rebuild on the entire index level so the value for fillfactor allows additional space at the leaf level for the whole index. If there is a local index, run reorg rebuild index at the partition level so only leaf pages in the local index partition are adjusted, leaving additional space for future inserts at the leaf level.

Note Adaptive Server 15.0 and later allows you to run reorg rebuild...index on local index partitions.

When you issue create index, the fillfactor value specified as part of the command is applied as follows:

- Clustered index:
  - On an allpages-locked table, the fillfactor is applied to the data pages.
  - On a data-only-locked table, the fillfactor is applied to the leaf pages of the index, and the data pages are fully packed (unless sp_chgattribute has been used to store a fillfactor for the table).
- Nonclustered index – the fillfactor value is applied to the leaf pages of the index.

You can also use sp_chgattribute to store values for fillfactor that are used when reorg rebuild is run on a table.

See "Setting fillfactor values" on page 57.

Advantages of using *fillfactor*

Setting fillfactor to a low value provides a temporary performance enhancement. As inserts to the database increase the amount of space used on data or index pages, its performance improvement decreases.

Using a lower value for fillfactor:

- Reduces page splits on the leaf-level of indexes, and the data pages of allpages-locked tables.
- Improves data-row clustering on data-only-locked tables with clustered indexes that experience inserts.

- Can reduce lock contention for tables that use page-level locking, since it reduces the likelihood that two processes will need the same data or index page simultaneously.
- Can help maintain large I/O efficiency for the data pages and for the leaf levels of nonclustered indexes, since page splits occur less frequently. This means that eight pages on an extent are likely to be sequential.

Disadvantages of using *fillfactor*

If you use fillfactor (especially with a very low value), you may notice these effects on queries and maintenance activities:

- More pages must be read for each query that performs a table scan or leaf-level scan on a nonclustered index.

  In some cases, a level may be added to an index's B-tree structure, since there will be more pages at the data level and possibly more pages at each index level.
- Increased index size, reducing the index's space efficiency. Because you cannot tune the fillfactor value at the page level, page splits with skewed data distribution occur frequently, even when there is available reserved space.
- dbcc commands take more time because they must check more pages.
- The time required to run dump database increases because more pages must be dumped. dump database copies all pages that store data, but does not dump pages that are not yet in use. Dumps and loads may also use more tapes.
- Fillfactor values fade over time. If you use fillfactor to reduce the performance impact of page splits, monitor your system and recreate indexes when page splitting begins to hurt performance.

Setting *fillfactor* values

Use sp_chgattribute to store a fillfactor percentage for each index and for the table. The fillfactor you set with sp_chgattribute is applied when you:

- Run reorg rebuild against tables using any locking scheme.
- Use alter table...partition by to repartition a table.

- Use alter table...lock to change the locking scheme for a table. or use an alter table...add/modify command that requires copying the table.
- Run create clustered index and a value is stored for the table.

See the *Reference Manual: Commands* for details information about each of these commands.

With the default fillfactor of 0, the index management process leaves room for two additional rows on each index page when you create a new index. When you set fillfactor to 100 percent, it no longer leaves room for these rows.The only effect that fillfactor has on size calculations is when calculating the number of clustered index pages and when calculating the number of non-leaf pages. Both of these calculations subtract 2 from the number of rows per page. Eliminate the -2 from these calculations.

Other values for fillfactor reduce the number of rows per page on data pages and leaf index pages. To compute the correct values when using fillfactor, multiply the size of the available data page (2016) by the fillfactor. For example, if your fillfactor is 75 percent, your data page would hold 1471 bytes. Use this value in place of 2016 when you calculate the number of rows per page. For these calculations, see "Compute the number of data pages" on page 91 and "Calculate the number of leaf pages in the index" on page 94.

Adaptive Server does not apply the stored fillfactor when it builds nonclustered indexes as a result of a create clustered index command:

- If a fillfactor value is specified with create clustered index, that value is applied to each nonclustered index.
- If no fillfactor value is specified with create clustered index, the server-wide default value (set with the default fillfactor percent configuration parameter) is applied to all indexes.

fillfactor examples

The following examples show the application of fillfactor values.

No stored *fillfactor* values

With no fillfactor values stored in sysindexes, Adaptive Server applies the fillfactor specified in create index as shown in Table 3-1.

```
create clustered index title_id_ix
on titles (title_id)
```

```
with fillfactor = 80
```

Table 3-1: fillfactor values applied with no table-level saved value

Command	Allpages-locked table	Data-only-locked table
create clustered index	Data pages: 80	Data pages: fully packed Leaf pages: 80
Nonclustered index rebuilds	Leaf pages: 80	Leaf pages: 80

The nonclustered indexes use the fillfactor specified in the create clustered index command.

If no fillfactor is specified in create clustered index, the nonclustered indexes always use the server-wide default; they never use a value from sysindexes.

Values used for *alter table...lock* and *reorg rebuild*

When no fillfactor values are stored, both alter table...lock and reorg rebuild apply the server-wide default value, set by default fillfactor percentage. The default fillfactor is applied as shown in Table 3-2.

Table 3-2: fillfactor values applied during rebuilds

Command	Allpages-locked table	Data-only-locked table
Clustered index rebuild	Data pages: default value	Data pages: fully packed Leaf pages: default value
Nonclustered index rebuilds	Leaf pages: default	Leaf pages: default

Table-level or clustered index *fillfactor* value stored

This command stores a fillfactor value of 50 for the table:

```
sp_chgattribute titles, "fillfactor", 50
```

If you set the stored table-level value for fillfactor to 50, this create clustered index command applies the fillfactor values shown in Table 3-3.

```
create clustered index title_id_ix
on titles (title_id)
with fillfactor = 80
```

Table 3-3: Using stored fillfactor values for clustered indexes

Command	Allpages-Locked Table	Data-Only-Locked Table
create clustered index	Data pages: 80	Data pages: 50 Leaf pages: 80
Nonclustered index rebuilds	Leaf pages: 80	Leaf pages: 80

Note When you run create clustered index, any table-level fillfactor value stored in sysindexes is reset to 0.

You must first issue sp_chgattribute to specify that data-only-locked data pages are filled during a create clustered index or reorg command.

Effects of *alter table...lock* when values are stored

Stored values for fillfactor are used when an alter table...lock command copies tables and rebuilds indexes.

Tables with clustered indexes

In an allpages-locked table, the table and the clustered index share the sysindexes row, so only one value for fillfactor can be stored and used for the table and clustered index. You can set the fillfactor value for the data pages by providing either the table name or the clustered index name. This command saves the value 50:

```
sp_chgattribute titles, "fillfactor", 50
```

This command saves the value 80, overwriting the value of 50 set by the previous command:

```
sp_chgattribute "titles.clust_ix", "fillfactor", 80
```

If you alter the titles table to use data-only locking after issuing the sp_chgattribute commands above, the stored value fillfactor of 80 is used for both the data pages and the leaf pages of the clustered index.

In a data-only-locked table, information about the clustered index is stored in a separate row in sysindexes. The fillfactor value you specify for the table applies to the data pages and the fillfactor value you specify for the clustered index applies to the leaf level of the clustered index.

When you change a DOL table to use allpages locking, the fillfactor stored for the table is used for the data pages. Adaptive Server ignores the fillfactor stored for the clustered index.

Table 3-4 shows the fillfactor values that are set on data and index pages using an alter table...lock command, executed after the sp_chgattribute commands above have been run.

Table 3-4: Effects of stored fillfactor values during alter table

alter table...lock	No clustered index	Clustered index
From allpages locking to data-only locking	Data pages: 80	Data pages: 80 Leaf pages: 80
From data-only locking to allpages locking	Data pages: 80	Data pages: 80

Note alter table...lock sets all stored fillfactor values for a table to 0.

fillfactor values stored for nonclustered indexes

Each nonclustered index is represented by a separate sysindexes row. These commands store different values for two nonclustered indexes:

```
sp_chgattribute "titles.ncl_ix", "fillfactor", 90
sp_chgattribute "titles.pubid_ix", "fillfactor", 75
```

Table 3-5 shows the effects of a reorg rebuild command on a data-only-locked table when the sp_chgattribute commands above are used to store fillfactor values.

Table 3-5: Effect of stored fillfactor values during reorg rebuild

reorg rebuild	No clustered index	Clustered index	Nonclustered indexes
Data-only-locked table	Data pages: 80	Data pages: 50 Leaf pages: 80	ncl_ix leaf pages: 90 pubid_ix leaf pages: 75

Using the *sorted_data* and *fillfactor* options

Use the sorted_data option for create index when the data to be sorted is already in an order specified by the index key. This allows create clustered index to skip data sorting, reallocating, and rebuilding the table's data pages..

For example, if data that is bulk copied into a table is already in order by the clustered index key, creating an index with the sorted_data option creates the index without performing a sort. If the data does not need to be copied to new pages, the fillfactor is not applied. However, the use of other create index options might still require copying.

See “Creating an index on sorted data” on page 107.

Reducing row forwarding

You may want to specify an expected row size for a data-only-locked table when an application allows rows with null values or short variable-length character fields to be inserted, and these rows grow in length with subsequent updates. Set an expected row size to reduce row forwarding.

For example, the titles table in the pubs2 database has many *varchar* columns and columns that allow null values. The maximum row size for this table is 331 bytes, and the average row size (as reported by optdiag) is 184 bytes, but you can insert a row with less than 40 bytes, since many columns allow null values. In a data-only-locked table, inserting short rows and then updating them may result in row forwarding.

See “Data-only-locked heap tables” on page 44.

Set the expected row size for tables with variable-length columns, using:

- exp_row_size parameter, in a create table statement.
- sp_chgattribute, for an existing table.
- A server-wide default value, using the configuration parameter default exp_row_size percent. This value is applied to all tables with variable-length columns, unless create table or sp_chgattribute is used to set a row size explicitly or to indicate that rows should be fully packed on data pages.

If you specify an expected row size value for an allpages-locked table, the value is stored in sysindexes, but the value is not applied during inserts and updates. If you later convert the table to data-only locking, Adaptive Server applies the exp_row_size during the conversion process and to all subsequent inserts and updates. The value for exp_row_size applies to the entire table.

Default, minimum, and maximum values for *exp_row_size*

Table 3-6 shows the minimum and maximum values for expected row size and the meaning of the special values 0 and 1.

Table 3-6: Valid values for expected row size

exp_row_size values	Minimum, maximum, and special values
Minimum	The greater of: • 2 bytes • The sum of all fixed-length columns

exp_row_size values	Minimum, maximum, and special values
Maximum	Maximum data row length
0	Use server-wide default value
1	Fully pack all pages; do not reserve room for expanding rows

You cannot specify an expected row size for tables that have fixed-length columns only. Columns that accept null values are, by definition, variable-length, since they are zero-length when null.

Default value

If you do not specify an expected row size or a value of 0 when you create a data-only-locked table with variable-length columns, Adaptive Server uses the amount of space specified by the configuration parameter default exp_row_size percent for any table that has variable-length columns.

See "Setting a default expected row size server-wide" on page 64 for information on how default exp_row_size affects space on data pages. Use sp_help to see the defined length of the columns in the table.

Specifying an expected row size with *create table*

This create table statement specifies an expected row size of 200 bytes:

```
create table new_titles (
      title_id    tid,
      title       varchar(80) not null,
      type        char(12),
      pub_id      char(4) null,
      price       money null,
      advance     money null,
      total_sales int null,
      notes       varchar(200) null,
      pubdate     datetime,
      contract    bit                             )
lock datapages
with exp_row_size = 200
```

Adding or changing an expected row size

Use sp_chgattribute to add or change the expected row size for a table. For example, to set the expected row size to 190 for the new_titles table, enter:

```
sp_chgattribute new_titles, "exp_row_size", 190
```

To switch the row size for a table from a current, explicit valut to the default exp_row_size percent, enter:

```
sp_chgattribute new_titles, "exp_row_size", 0
```

To fully pack the pages, rather than saving space for expanding rows, set the value to 1.

Changing the expected row size with sp_chgattribute does not immediately affect the storage of existing data. The new value is applied:

- When you create a clustered index on the table or run reorg rebuild. The expected row size is applied as rows are copied to new data pages.

  If you increase exp_row_size, and recreate the clustered index or run reorg rebuild, the new copy of the table may require more storage space.

- The next time a page is affected by data modifications.

Setting a default expected row size server-wide

default exp_row_size percent reserves a percentage of the page size to set aside for expanding updates. The default value, 5, sets aside 5% of the space available per data page for all data-only-locked tables that include variable-length columns. Since there are 2002 bytes available on data pages in data-only-locked tables, the default value sets aside 100 bytes for row expansion. This command sets the default value to 10%:

```
sp_configure "default exp_row_size percent", 10
```

Setting default exp_row_size percent to 0 means that no space is reserved for expanding updates for any tables where the expected row size is not explicitly set with create table or sp_chgattribute.

If an expected row size for a table is specified with create table or sp_chgattribute, that value takes precedence over the server-wide setting.

Displaying the expected row size for a table

Use sp_help to display the expected row size for a table:

```
sp_help titles
```

If the value is 0, and the table has nullable or variable-length columns, use sp_configure to display the server-wide default value:

```
sp_configure "default exp_row_size percent"
```

This query displays the value of the exp_rowsize column for all user tables in a database:

```
select object_name(id), exp_rowsize
from sysindexes
where id > 100 and (indid = 0 or indid = 1)
```

Choosing an expected row size for a table

Setting an expected row size helps reduce the number of forwarded rows only if the rows expand after they are inserted into the table. Setting the expected row size correctly means that:

- Your application results in a small percentage of forwarded rows.
- You do not waste space on data pages due to over-allocating space towards the expected row size value.

Using *optdiag* to check for forwarded rows

For tables that already contain data, use optdiag to display statistics for the table. The "Data row size" shows the average data row length, including the row overhead. This sample optdiag output for the titles table shows 12 forwarded rows and an average data row size of 184 bytes:

```
Statistics for table:                "titles"

     Data page count:                655
     Empty data page count:          5
     Data row count:                 4959.000000000
     Forwarded row count:            12.000000000
     Deleted row count:              84.000000000
     Data page CR count:             0.000000000
     OAM + allocation page count:    6
     Pages in allocation extent:     1
     Data row size:                  184.000000000
```

Use optdiag to check the number of forwarded rows for a table to determine whether your setting for exp_row_size is reducing the number of forwarded rows generated by your applications.

See Chapter 2, "Statistics Tables and Displaying Statistics with optdiag," in the *Performance and Tuning Series: Improving Performance with Statistical Analysis*.

Querying *systabstats* for forwarded rows

The forwrowcnt column in the systabstats table stores the number of forwarded rows for a table. To display the number of forwarded rows and average row size for all user tables with object IDs greater than 100, use this query:

```
select objectname = object_name(id),
        partitionname = (select name from syspartitions p
          where p.id = t.id and p.indid = t.indid)
      , forwrowcnt, datarowsize
      , exprowsize = (select i.exp_rowsize from sysindexes i
         where i.id = t.id and i.indid = t.indid)
into #temptable
from systabstats t
where id > 100 and indid IN (0,1)

exec sp_autoformat #temptable
```

Note Forwarded row counts are updated in memory, and the housekeeper tasks periodically flushes them to disk.

Query the systabstats table using SQL, use sp_flushstats first to ensure that the most recent statistics are available. optdiag flushes statistics to disk before displaying values.

Conversion of *max_rows_per_page* to *exp_row_size*

If a max_rows_per_page value is set for an allpages-locked table, the value is used to compute an expected row size during the alter table...lock command. The formula is shown in Table 3-7.

Table 3-7: Conversion of max_rows_per_page to exp_row_size

Value of max_rows_per_page	Value of exp_row_size
0	Percentage value set by default exp_row_size percent

Value of max_rows_per_page	Value of exp_row_size
1 – 254	The smaller of: • Maximum row size • (logical page size) – (page header overheads) / max_rows_per_page

For example, if max_rows_per_page is set to 10 for an allpages-locked table on a server configured for 2K pages with a maximum defined row size of 300 bytes, the exp_row_size value is 200 (2002/10) after the table is altered to use data-only locking.

If max_rows_per_page is set to 10, but the maximum defined row size is only 150, the expected row size value is set to 150.

Monitoring and managing tables that use expected row size

After setting an expected row size for a table, use optdiag or queries on systabstats to determine the number of forwarded rows being generated by your applications. Run reorg forwarded_rows the number of forwarded rows is high enough to affect application performance. reorg forwarded_rows uses short transactions and is nonintrusive, so you can run it while applications are active.

See Chapter 9 “Using the reorg Command,” in the *System Administration Guide: Volume 2*.

You can monitor forwarded rows on a per-partition basis, and run reorg forwarded rows on those partitions that have a large number of forwarded rows. See the *Reference Manual: Commands*.

If the application continues to generate a large number of forwarded rows, consider using sp_chgattribute to increase the expected row size for the table.

You may want to allow a certain percentage of forwarded rows. If running reorg to clear forwarded rows does not cause concurrency problems for your applications, or if you can run reorg at nonpeak times, allowing a small percentage of forwarded rows does not cause a serious performance problem.

Setting the expected row size for a table increases the amount of storage space and the number of I/Os required to read a set of rows. If the increase in the number of I/Os due to increased storage space is high, allowing rows to be forwarded and occasionally running reorg may have less overall performance impact.

Leaving space for forwarded rows and inserts

Set a reservepagegap value to reduce storage fragmentation, thus also reducing the frequency of maintenance activities such as running reorg rebuild and recreating indexes for some tables. Good performance on data-only-locked tables requires good data clustering on the pages, extents, and allocation units used by the table.

The clustering of data and index pages in physical storage stays high as long as there is space nearby for storing forwarded rows and rows that are inserted in index key order. Use the reservepagegap space management property to reserve empty pages for expansion when additional pages need to be allocated.

Row and page cluster ratios are usually 1.0, or very close to 1.0, immediately after you create a clustered index on a table or immediately after you run reorg rebuild. However, future data modifications may cause row forwarding and require allocation of additional data and index pages to store inserted rows.

You can set the reserve page gap on the data and index layer pages for allpages and data-only-locked tables.

Extent allocation commands and *reservepagegap*

Extent allocation means that pages are allocated in multiples of eight, rather than one page at a time. This reduces logging activity by writing only one log record instead of eight.

Commands that perform extent allocation are: select into, create index, reorg rebuild, bcp, alter table...lock, and the alter table...unique and primary key constraint options, since these constraints create indexes. alter table commands that add, drop, or modify columns, or change a table's partitioning scheme sometimes also require a table-copy operation. By default, all these commands use extent allocation.

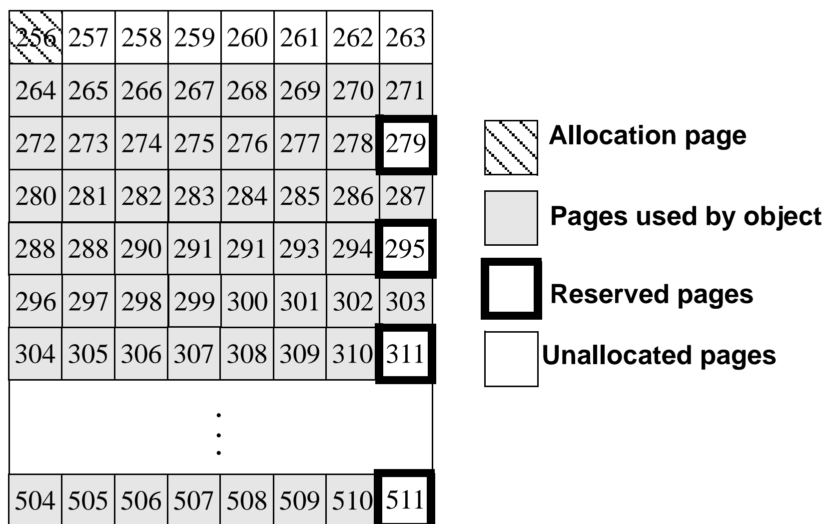
Specify reservepagegap value in pages, indicating a ratio of empty pages to filled pages. For example, if you specify a reservepagegap value of 8, an operation that uses extent allocation fills seven pages and leaves the eighth page empty.

Extent allocation operations do not use the first page on each allocation unit, because it stores the allocation page. For example, if you create a clustered index on a large table and do not specify a reserve page gap, each allocation unit has seven empty, unallocated pages, 248 used pages, and the allocation page. Adaptive Server can use the seven empty pages for row forwarding and inserts to the table, which helps keep forwarded rows and inserts with clustered indexes on the same allocation unit. Using reservepagegap leaves additional empty pages on each allocation unit.

See Chapter 12, "Creating Indexes on Tables" in the *Transact-SQL Users Guide* for information about when to use reservepagegap.

Figure 3-1 shows how an allocation unit might look after a clustered index is created with a reservepagegap value of 16 on the table. The pages that share the first extent with the allocation unit are not used and are not allocated to the table. Pages 279, 295, and 311 are the unused pages on extents that are allocated to the table.

Figure 3-1: Reserved pages after creating a clustered index



Specifying a reserve page gap with *create table*

This create table command specifies a reservepagegap value of 16:

```
create table more_titles (
      title_id    tid,
      title       varchar(80) not null,
      type        char(12),
      pub_id      char(4) null,
      price       money null,
      advance     money null,
      total_sales int null,
      notes       varchar(200) null,
      pubdate     datetime,
      contract    bit
    )
lock datarows
with reservepagegap = 16
```

Any operation that performs extent allocation on the more_titles table leaves 1 empty page for each 15 filled pages. For partitioned tables, the reservepagegap value applies to all partitions.

The default value for reservepagegap is 0, meaning that no space is reserved.

Specifying a reserve page gap with *create index*

This command specifies a reservepagegap of 10 for nonclustered index pages:

```
create index type_price_ix
on more_titles(type, price)
with reservepagegap = 10
```

You can specify a reservepagegap value with the alter table...constraint options, primary key and unique, that create indexes. The value of reservepagegap for local index on partitioned tables applies to all local index partitions.

This example creates a unique constraint:

```
alter table more_titles
add constraint uniq_id unique (title_id)
with reservepagegap = 20
```

Changing *reservepagegap*

To change the reserve page gap for the titles table to 20, enter:

```
sp_chgattribute more_titles, "reservepagegap", 20
```

This command sets the reserve page gap for the index title_ix to 10:

```
sp_chgattribute "titles.title_ix",
    "reservepagegap", 10
```

sp_chgattribute changes only values in system tables; data is not moved on data pages as a result of running the procedure. Changing reservepagegap for a table affects future storage as follows:

- When data is bulk-copied into the table, the reserve page gap is applied to all newly allocated space, but the storage of existing pages is not affected.
- Any command that copies the table's data to create a new version of the table applies the reserve page gap during the data copy phase of the operation. For example, using reorg rebuild or using alter table to change the locking or partitioning scheme of a table or any change of schema that requires a data copy both apply to reserver page gap.
- When you create a clustered index, the reserve page gap value stored for the table is applied to the data pages.

The reserve page gap is applied to index pages during:

- alter table...lock, indexes are rebuilt.
- The index rebuild phase during reorg rebuild when using alter table to change the locking or partitioning scheme of a table, or when changing any schema that requires a data copy.
- create clustered index and alter table commands that create a clustered index, as nonclustered indexes are rebuilt

reservepagegap examples

These examples show how reservepagegap is applied during alter table and reorg rebuild commands.

reservepagegap specified only for the table

The following commands specify a reservepagegap for the table, but do not specify a value in the create index commands:

```
sp_chgattribute titles, "reservepagegap", 16
create clustered index title_ix on titles(title_id)
create index type_price on titles(type, price)
```

Table 3-8 shows the values applied when running reorg rebuild or dropping and creating a clustered index.

Table 3-8: reservepagegap values applied with table-level saved value

Command	Allpages-locked table	Data-only-locked table
create clustered index or clustered index rebuild due to reorg rebuild	Data and index pages: 16	Data pages: 16 Index pages: 0 (filled extents)
Nonclustered index rebuild	Index pages: 0 (filled extents)	Index pages: 0 (filled extents)

For an allpages-locked table with a clustered index, reservepagegap is applied to both the data and index pages. For a data-only-locked table, reservepagegap is applied to the data pages, but not to the clustered index pages.

reservepagegap specified for a clustered index

These commands specify different reservepagegap values for the table and the clustered index, and a value for the nonclustered type_price index:

```
sp_chgattribute titles, "reservepagegap", 16
create clustered index title_ix on titles(title)
    with reservepagegap = 20
create index type_price on titles(type, price)
    with reservepagegap = 24
```

Table 3-9 shows the effects of this sequence of commands.

Table 3-9: reservepagegap values applied with for index pages

Command	Allpages-locked table	Data-only-locked table
create clustered index or clustered index rebuild due to reorg rebuild	Data and index pages: 20	Data pages: 16 Index pages: 20
Nonclustered index rebuilds	Index pages: 24	Index pages: 24

For allpages-locked tables, the reservepagegap specified with create clustered index applies to both data and index pages. For data-only-locked tables, the reservepagegap specified with create clustered index applies only to the index pages. If there is a stored reservepagegap value for the table, that value is applied to the data pages.

Choosing a value for *reservepagegap*

Choosing a value for reservepagegap depends on:

- Whether the table has a clustered index,
- The rate of inserts to the table,

- The number of forwarded rows that occur in the table, and
- The frequency with which you recreate the clustered index or run the reorg rebuild command.

When reservepagegap is configured correctly, enough pages are left for allocation of new pages to tables and indexes so that the cluster ratios for the table, clustered index, and nonclustered leaf-level pages remain high during the intervals between regular index maintenance tasks.

Monitoring *reservepagegap* settings

Use optdiag to check the cluster ratio and the number of forwarded rows in tables. Declines in cluster ratios may also indicate that you can improve performance by running reorg commands:

- If the data page cluster ratio for a clustered index is low, run reorg rebuild or drop and recreate the clustered index.
- If the index page cluster ratio is low, drop and recreate the nonclustered index.

To reduce the frequency with which you run reorg commands to maintain cluster ratios, increase the reservepagegap slightly before running reorg rebuild.

See Chapter 2, “Statistics Tables and Displaying Statistics with optdiag,” in *Performance and Tuning Series: Improving Performance with Statistical Analysis*.

reservepagegap and *sorted_data* options

When you create a clustered index on a table that is already stored on the data pages in index key order, the sorted_data option suppresses the step of copying the data pages in key order for unpartitioned tables. The reservepagegap option can be specified in create clustered index commands, to leave empty pages on the extents used by the table, leaving room for later expansion. There are rules that determine which option takes effect. You cannot use sp_chgattribute to change the reservepagegap value and get the benefits of both of these options.

If you specify both with create clustered index:

- On unpartitioned, allpages-locked tables, if the reservepagegap value specified with create clustered index matches the values already stored in sysindexes, the sorted_data option takes precedence. Data pages are not copied, so the reservepagegap is not applied. If the reservepagegap value specified in the create clustered index command is different from the values stored in sysindexes, the data pages are copied, and the reservepagegap value specified in the command is applied to the copied pages.
- On data-only-locked tables, the reservepagegap value specified with create clustered index applies only to the index pages. Data pages are not copied.

Besides reservepagegap, other options to create clustered index may require a sort, which causes the sorted_data option to be ignored. For more information, see “Creating an index on sorted data” on page 107.

In particular, the following comments relate to the use of reservepagegap:

- On partitioned tables, any create clustered index command that requires copying data pages performs a parallel sort and then copies the data pages in sorted order, applying the reservepagegap values as the pages are copied to new extents.
- Whenever the sorted_data option is not superseded by other create clustered index options, the table is scanned to determine whether the data is stored in key order. The index is built during the scan, without a sort being performed.

Table 3-10 shows how these rules apply.

Table 3-10: reservepagegap and sorted_data options

	Partitioned table	Unpartitioned table
Allpages-locked table		
create index with sorted_data and matching reservepagegap value	Does not copy data pages; builds the index as pages are scanned.	Does not copy data pages; builds the index as pages are scanned.
create index with sorted_data and different reservepagegap value	Performs parallel sort, applying reservepagegap as pages are stored in new locations in sorted order.	Copies data pages, applying reservepagegap and building the index as pages are copied; no sort is performed.
Data-only-locked table		
create index with sorted_data and any reservepagegap value	reservepagegap applies to index pages only; does not copy data pages.	reservepagegap applies to index pages only; does not copy data pages.

Matching options and goals

To redistribute the data pages of a table, leaving room for later expansion:

- For allpages-locked tables, drop and recreate the clustered index without using the sorted_data option. If the value stored in sysindexes is not the value you want, use create clustered index to specify the desired reservepagegap.
- For data-only-locked tables, use sp_chgattribute to set the reservepagegap for the table to the desired value, then drop and recreate the clustered index, without using the sorted_data option. The reservepagegap stored for the table applies to the data pages. If reservepagegap is specified in the create clustered index command, it applies only to the index pages.

To create a clustered index without copying data pages:

- For allpages-locked tables, use the sorted_data option, but do not use create clustered index to specify a reservepagegap. Alternatively, specify a value that matches the value stored in sysindexes.
- For data-only-locked tables, use the sorted_data option. If a reservepagegap value is specified in the create clustered index command, it applies only to the index pages and does not cause data page copying.

To use the sorted_data option following a bulk-copy operation, a select into command, or another command that uses extent allocation, set the reservepagegap value that you want for the data pages before copying the data, or specify it in the select into command. Once the data pages have been allocated and filled, the following command applies reservepagegap to the index pages only, since the data pages do not need to be copied:

```
create clustered index title_ix
on titles(title_id)
with sorted_data, reservepagegap = 32
```

Using *max_rows_per_page* on allpages-locked tables

Setting a maximum number of rows per pages can reduce contention for allpages-locked tables and indexes. In most cases, it is preferable to convert the tables to use a data-only-locking scheme. If there is some reason that you cannot change the locking scheme, and contention is a problem on an allpages-locked table or index, setting a max_rows_per_page value may help performance.

When there are fewer rows on the index and data pages, the chances of lock contention are reduced. As the keys are spread out over more pages, it becomes more likely that the page you want is not the page someone else needs. To change the number of rows per page, adjust the fillfactor or max_rows_per_page values of your tables and indexes.

fillfactor (defined by either sp_configure or create index) determines how full Adaptive Server makes each data page when it creates a new index on existing data. Since fillfactor helps reduce page splits, exclusive locks are also minimized on the index, improving performance. However, the fillfactor value is not maintained by subsequent changes to the data. max_rows_per_page (defined by sp_chgattribute, create index, create table, or alter table) is similar to fillfactor, except that Adaptive Server maintains the max_rows_per_page value as the data changes.

The costs associated with decreasing the number of rows per page using fillfactor or max_rows_per_page include more I/O to read the same number of data pages, more memory for the same performance from the data cache, and more locks. In addition, a low value for max_rows_per_page for a table may increase page splits when data is inserted into the table.

Reducing lock contention

The max_rows_per_page value specified in a create table, create index, or alter table command restricts the number of rows allowed on a data page, a clustered index leaf page, or a nonclustered index leaf page. This reduces lock contention and improves concurrency for frequently accessed tables.

max_rows_per_page applies to the data pages of a heap table, or the leaf pages of an index. Unlike fillfactor, which is not maintained after creating a table or index, Adaptive Server retains the max_rows_per_page value when adding or deleting rows.

The following command creates the sales table and limits the maximum rows per page to four:

```
create table sales
      (stor_id          char(4)       not null,
      ord_num           varchar(20)   not null,
      date              datetime      not null)
      with max_rows_per_page = 4
```

If you create a table with a max_rows_per_page value, and then create a clustered index on the table without specifying max_rows_per_page, the clustered index inherits the max_rows_per_page value from the create table statement. Creating a clustered index with max_rows_per_page changes the value for the table's data pages.

Indexes and *max_rows_per_page*

The default value for max_rows_per_page is 0, which creates clustered indexes with full data pages, creates nonclustered indexes with full leaf pages, and leaves a comfortable amount of space within the index B-tree in both the clustered and nonclustered indexes.

For heap tables and clustered indexes, the range for max_rows_per_page is 0 – 256.

For nonclustered indexes, the maximum value for max_rows_per_page is the number of index rows that fit on the leaf page, without exceeding 256. To determine the maximum value, subtract 32 (the size of the page header) from the page size and divide the difference by the index key size. The following statement calculates the maximum value of max_rows_per_page for a nonclustered index:

```
select (@@pagesize - 32)/minlen
    from sysindexes
    where name = "indexname"
```

select into and *max_rows_per_page*

By default, select into does not carry over a base table's max_rows_per_page value, but creates the new table with a max_rows_per_page value of 0. However, you can add the with max_rows_per_page option to select into to specify a value other than 0.

Applying *max_rows_per_page* to existing data

There are several ways to apply a max_rows_per_page value to existing data:

- If the table has a clustered index, drop and recreate the index using a different max_rows_per_page value.

- Use sp_chgattribute to change the value of max_rows_per_page, then rebuild the entire table and its indexes with reorg rebuild. For example, to change the max_rows_per_page value of the authors table to 1, enter:

```
sp_chgattribute authors, "max_rows_per_page", 1
go
reorg rebuild authors
go
```

- Use bcp to repopulate the table, and:

  a Copy out the table data.

  b Truncate the table.

  c Use sp_chgattribute to set the max_rows_per_page value.

  d Copy the data back in.

CHAPTER 4 Table and Index Size

This chapter explains how to determine the current sizes of tables and indexes and how to estimate table size for space planning.

Topic	Page
Determining the sizes of tables and indexes	80
Effects of data modifications on object sizes	81
Using optdiag to display object sizes	81
Using sp_spaceused to display object size	82
Using sp_estspace to estimate object size	84
Using formulas to estimate object size	86

Knowing the sizes of your tables and indexes is important to understanding query and system behavior. At several stages of tuning work, you need size data to:

- Understand statistics io reports for a specific query plan. Chapter 1, "Using the set statistics Commands," in *Performance and Tuning Series: Improving Performance with Statistical Analysis* describes how to use statistics io to examine the I/O performed.
- Understand the optimizer's choice of query plan. The Adaptive Server cost-based optimizer estimates the physical and logical I/O required for each possible access method and chooses the cheapest method. If you think a particular query plan is unusual, use dbcc traceon(302) to determine why the optimizer made the decision. This output includes page number estimates.
- Determine object placement, based on the sizes of database objects and the expected I/O patterns on the objects. Iimprove performance by distributing database objects across physical devices so that reads and writes to disk are evenly distributed. Object placement is described in Chapter 1, "Controlling Physical Data Placement."

- Understand changes in performance. If objects grow, their performance characteristics can change. One example is a table that is heavily used and is usually 100% cached. If that table grows too large for its cache, queries that access the table can suddenly suffer poor performance. This is particularly true for joins requiring multiple scans.
- Perform capacity planning. Whether you are designing a new system or planning for growth of an existing system, you must know your space requirements to plan for physical disks and memory needs.
- Understand output from Adaptive Server Monitor and from sp_sysmon reports on physical I/O.

Determining the sizes of tables and indexes

Adaptive Server includes several tools that provide information about the current sizes of tables or indexes, or that can predict future sizes:

- optdiag displays the sizes and many other statistics for tables and indexes. See Chapter 2, "Statistics Tables and Displaying Statistics with optdiag," in *Performance and Tuning Series: Improving Performance with Statistical Analysis*
- sp_spaceused reports on the current size of existing tables and indexes.
- sp_estspace can predict the size of a table and its indexes, given a number of rows as a parameter.

You can also compute table and index size using formulas provided in this chapter. sp_spaceused and optdiag report actual space usage. The other methods presented in this chapter provide size estimates.

For partitioned tables, sp_helppartition reports on the number of pages stored on each partition of the table. See Chapter 10, "Partitioning Tables and Indexes" in the *Transact-SQL Users Guide*.

Effects of data modifications on object sizes

Over time, the effects of randomly distributed data modifications on a set of tables tend to produce data pages and index pages that average approximately 75% full. The major factors are:

- When you insert a row to be placed on a page of an allpages-locked table with a clustered index, and there is no room on the page for that row, the page is split, leaving two pages that are about 50 percent full.
- When you delete rows from heaps or from tables with clustered indexes, the space used on the page decreases. You can have pages that contain very few rows or even a single row.
- After some deletes or page splits have occurred, inserting rows into tables with clustered indexes tends to fill up pages that have been split, or pages where rows have been deleted.

Page splits also take place when rows need to be inserted into full index pages, so index pages also tend to average approximately 75% full, unless you drop and recreate them periodically.

Using *optdiag* to display object sizes

The optdiag command displays statistics for tables, indexes, and columns, including the size of tables and indexes. If you are performing query tuning, optdiag provides the best tool for viewing all the statistics you need. Here is a sample report for the titles table in the pubtune database:

```
Table owner:                       "dbo"
Statistics for table:              "titles"
     Data page count:              662
     Empty data page count:        10
     Data row count:               4986.0000000000000000
     Forwarded row count:          18.0000000000000000
     Deleted row count:            87.0000000000000000
     Data page CR count:           86.0000000000000000
     OAM + allocation page count:  5
     First extent data pages:      3
Data row size:                238.8634175691937287
```

See Chapter 2, “Statistics Tables and Displaying Statistics with optdiag,” in *Performance and Tuning Series: Improving Performance with Statistical Analysis*.

Advantages of *optdiag*

The advantages of optdiag are that:

- It can display statistics for all tables in a database, or for a single table.
- optdiag output contains addition information useful for understanding query costs, such as index height and the average row length.
- It is frequently used for other tuning tasks, so you may have these reports readily available.

Disadvantages of *optdiag*

The principle disadvantage of optdiag is that it produces a lot of output. If you need only a single piece of information, such as the number of pages in a table, other methods are faster and incur lower system overhead.

Using *sp_spaceused* to display object size

The system procedure sp_spaceused reads values stored on an object's OAM page to provide a quick report on the space used by the object.

```
                sp_spaceused titles
name         rowtotal reserved   data      index_size  unused
------------ -------- ---------- --------- ----------- --------
titles       5000        1756 KB   1242 KB     440 KB     74 KB
```

The rowtotal value may be inaccurate at times; not all Adaptive Server processes update this value on the OAM page. The commands update statistics, dbcc checktable, and dbcc checkdb correct the rowtotal value on the OAM page. Table 4-1 explains the headings in sp_spaceused output.

Table 4-1: sp_spaceused output

Column	Meaning
rowtotal	Reports an estimate of the number of rows. The value is read from the OAM page. Though not always exact, this estimate is much quicker and leads to less contention than select count(*).
reserved	Reports pages reserved for use by the table and its indexes. It includes both the used and unused pages in extents allocated to the objects. It is the sum of data, index_size, and unused.
data	Reports the kilobytes on pages used by the table.
index_size	Reports the total kilobytes on pages used by the indexes.
unused	Reports the kilobytes of unused pages in extents allocated to the object, including the unused pages for the object's indexes.

To report index sizes separately, use:

```
                     sp_spaceused titles, 1
 index_name           size       reserved   unused
 -------------------- ---------- ---------- ----------
 title_id_cix         14 KB      1294 KB    38 KB
 title_ix             256 KB     272 KB     16 KB
 type_price_ix        170 KB     190 KB     20 KB

name         rowtotal reserved   data      index_size  unused
------------ -------- ---------- --------- ----------- --------
titles       5000        1756 KB   1242 KB     440 KB     74 KB
```

For clustered indexes on allpages-locked tables, the size value represents the space used for the root and intermediate index pages. The reserved value includes the index size and the reserved and used data pages.

The "1" in the sp_spaceused syntax indicates that detailed index information should be printed. It has no relation to index IDs or other information.

Advantages of *sp_spaceused*

The advantages of sp_spaceused are that:

- It provides quick reports without excessive I/O and locking, since it uses only values in the table and index OAM pages to return results.

- It shows the amount of space that is reserved for expansion of the object, but not currently used to store data.
- It provides detailed reports on the size of indexes and of text and image, and Java off-row column storage.

Note Use sp_helpartition to report the number of pages in each partition. sp_helpartition does not report the same level of detail as sp_spaceused, but does give a general idea of the amount of space a partition uses. In Adaptive Server version 15.0.2 and later, sp_spaceusage provides detailed information about a variety of subjects, including the space used by tables at the index and partition level, and fragmentation.

See the *Adaptive Server Reference Manual: Procedures* for more information about all these system procedures.

Disadvantages of *sp_spaceused*

The disadvantages of sp_spaceused are:

- It may report inaccurate counts for row total and space usage.
- Output is only in kilobytes, while most query-tuning activities use pages as a unit of measure. However, you can use sp_spaceusage to report information in any unit you specify.

Using *sp_estspace* to estimate object size

sp_spaceused and optdiag report on actual space usage. sp_estspace can help you plan for future growth of your tables and indexes. This procedure uses information in the system tables (sysobjects, syscolumns, and sysindexes) to determine the length of data and index rows. You provide a table name, and the number of rows you expect to have in the table, and sp_estspace estimates the size for the table and for any indexes that exist. It does not look at the actual size of the data in the tables.

To use sp_estspace:

- Create the table, if it does not already exist.

- Create any indexes on the table.
- Execute the procedure, estimating the number of rows that the table will hold.

The output reports the number of pages and bytes for the table and for each level of the index.

The following example estimates the size of the titles table with 500,000 rows, a clustered index, and two nonclustered indexes:

```
sp_estspace titles, 500000
name                  type         idx_level Pages    Kbytes
--------------------- ------------ --------- -------- --------
titles                data                 0    50002   100004
title_id_cix          clustered            0      302      604
title_id_cix          clustered            1        3        6
title_id_cix          clustered            2        1        2
title_ix              nonclustered         0    13890    27780
title_ix              nonclustered         1      410      819
title_ix              nonclustered         2       13       26
title_ix              nonclustered         3        1        2
type_price_ix         nonclustered         0     6099    12197
type_price_ix         nonclustered         1       88      176
type_price_ix         nonclustered         2        2        5
type_price_ix         nonclustered         3        1        2

Total_Mbytes
-----------------
  138.30

name                  type         total_pages  time_mins
--------------------- ------------ ------------ ------------
title_id_cix          clustered           50308          250
title_ix              nonclustered        14314           91
type_price_ix         nonclustered         6190           55
```

sp_estspace also allows you to specify a fillfactor, the average size of variable-length fields and text fields, and the I/O speed. For more information, see *Reference Manual: Procedures*.

Note The index creation times printed by sp_estspace do not factor in the effects of parallel sorting.

Advantages of *sp_estspace*

The advantages of sp_estspace are that it:

- Provides an efficient way to perform initial capacity planning and to plan for table and index growth.
- Helps you estimate the number of index levels.
- Helps you estimate future disk space, cache space, and memory requirements.

Disadvantages of *sp_estspace*

The disadvantages of sp_estspace are that:

- Returned sizes are only estimates and may differ from actual sizes, due to fillfactors, page splitting, actual size of variable-length fields, and other factors.
- Index creation times can vary widely, depending on disk speed, the use of extent I/O buffers, and system load.

Using formulas to estimate object size

The formulas discussed here can help you estimate the future sizes of the tables and indexes in your database. The amount of overhead in each row for tables and indexes that contain variable-length fields is greater than the overhead for tables that contain only fixed-length fields, so two sets of formulas are required.

The process involves calculating the number of bytes of data and overhead for each row, and dividing that number into the number of bytes available on a data page. Each page requires some overhead, which limits the number of bytes available for data:

- For allpages-locked tables, page overhead is 32 bytes, leaving 2016 bytes available for data on a 2K page.
- For data-only-locked tables, page overhead is 46 bytes, leaving 2002 bytes available for data.

For the most accurate estimate, round down divisions that calculate the number of rows per page (rows are never split across pages), and round up divisions that calculate the number of pages.

Factors that can affect storage size

Using space management properties can increase the space needed for a table or an index. See "Effects of space management properties" on page 101, and "max_rows_per_page" on page 102.

If your table includes text or image datatypes or Java off-row columns, use 16 (the size of the text pointer that is stored in the row) in your calculations. Then see "LOB pages" on page 103 to see how to calculate the storage space required for the actual text or image data.

Indexes on data-only-locked tables may be smaller than the formulas predict due to two factors:

- Duplicate keys are stored only once, followed by a list of row IDs for the key.
- Compression of keys on nonleaf levels; only enough of the key to differentiate from the neighboring keys is stored. This is especially effective in reducing the size when long character keys are used.

If the configuration parameter page utilization percent is set to less than 100, Adaptive Server may allocate new extents before filling all pages on the allocated extents. This does not change the number of pages used by an object, but leaves empty pages in the extents allocated to the object.

Storage sizes for datatypes

The storage sizes for datatypes are shown in Table 4-2:

Table 4-2: Storage sizes for Adaptive Server datatypes

Datatype	Size
char	Defined size
nchar	Defined size * *@@ncharsize*
unichar	n**@@unicharsize (@@unicharsize equals 2)*
univarchar	the actual number of characters**@@unicharsize*
varchar	Actual number of characters
nvarchar	Actual number of characters * *@@ncharsize*
binary	Defined size
varbinary	Data size
int	4
smallint	2
tinyint	1
float	4 or 8, depending on precision
double precision	8
real	4
numeric	2–17, depending on precision and scale
decimal	2–17, depending on precision and scale
money	8
smallmoney	4
datetime	8
smalldatetime	4
bit	1
text	16 bytes + 2K * number of pages used
image	16 bytes + 2K * number of pages used
timestamp	8

The storage size for a numeric or decimal column depends on its precision. The minimum storage requirement is 2 bytes for a 1- or 2-digit column. Storage size increases by 1 byte for each additional 2 digits of precision, up to a maximum of 17 bytes.

Any columns defined as NULL are considered variable-length columns, since they involve the overhead associated with variable-length columns.

All calculations in the examples that follow are based on the maximum size for varchar, univarchar, nvarchar, and varbinary data—the defined size of the columns. They also assume that the columns were defined as NOT NULL.

Tables and indexes used in the formulas

The example illustrates the computations on a table that contains 9,000,000 rows:

- The sum of fixed-length column sizes is 100 bytes.
- The sum of variable-length column sizes is 50 bytes; there are 2 variable-length columns.

The table has two indexes:

- A clustered index, on a fixed-length column, of 4 bytes
- A composite nonclustered index with these columns:
    - A fixed length column, of 4 bytes
    - A variable length column, of 20 bytes

Different formulas are needed for allpages-locked and data-only-locked tables, since they have different amounts of overhead on the page and per row:

- See "Calculating table and clustered index sizes for allpages-locked tables" on page 89 for tables that use allpages-locking.
- See "Calculating the sizes of data-only-locked tables" on page 96 for the formulas to use if tables that use data-only locking.

Calculating table and clustered index sizes for allpages-locked tables

The formulas and examples for allpages-locked tables are listed below as a series of steps. Steps 1–6 outline the calculations for an allpages-locked table with a clustered index, giving the table size and the size of the index tree. Steps 7–12 outline the calculations for computing the space required by nonclustered indexes. All of the formulas use the maximum size of the variable-length fields. The steps are:

1. "Calculate the data row size" on page 90
2. "Compute the number of data pages" on page 91
3. "Compute the size of clustered index rows" on page 91
4. "Compute the number of clustered index pages" on page 92
5. "Compute the total number of index pages" on page 92

6 "Calculate allocation overhead and total pages" on page 93
7 "Calculate the size of the leaf index row" on page 94
8 "Calculate the number of leaf pages in the index" on page 94
9 "Calculate the size of the nonleaf rows" on page 95
10 "Calculate the number of non-leaf pages" on page 95
11 "Calculate the total number of non-leaf index pages" on page 95
12 "Calculate allocation overhead and total pages" on page 96

These formulas show how to calculate the sizes of tables and clustered indexes. If your table does not have clustered indexes, skip steps 3, 4, and 5. When you have computed the number of data pages in step 2, go to step 6 to add the number of OAM pages.

optdiag output includes the average length of data rows and index rows. You can use these values for the data and index row lengths, if you want to use average lengths instead.

Calculate the data row size

Rows that store variable-length data require more overhead than rows that contain only fixed-length data, so there are two separate formulas for computing the size of a data row.

Fixed-length columns only

If the table contains only fixed-length columns, and all are defined as NOT NULL, use:

Formula

4	(Overhead)
+	Sum of bytes in all fixed-length columns
	= Data row size

Some variable-length columns

If the table contains any variable-length columns or columns that allow NULL values, use this formula.

The table in the example contains variable-length columns, so the computations are shown in the right column.

Formula		Example
4	(Overhead)	4
+	Sum of bytes in all fixed-length columns	+ 100
+	Sum of bytes in all variable-length columns	+ 50
	= Subtotal	154
+	(Subtotal / 256) + 1 (Overhead)	1
+	Number of variable-length columns + 1	3
+ 2	(Overhead)	2
	= Data row size	160

Compute the number of data pages

Formula

2016 / Data row size = Number of data rows per page

Number of rows / Rows per page = Number of data pages required

Example

2016 / 160	=	12 data rows per page
9,000,000 / 12	=	750,000 data pages

Compute the size of clustered index rows

Index rows containing variable-length columns require more overhead than index rows containing only fixed-length values. Use the first formula if all the keys are fixed length. Use the second formula if the keys include variable-length columns or allow NULL values.

Fixed-length columns only

The clustered index in the example has only fixed-length keys.

Formula		Example
5	(Overhead)	5
+	Sum of bytes in the fixed-length index keys	+ 4
	= Clustered row size	9

Some variable-length columns

	5	(Overhead)
+		Sum of bytes in the fixed-length index keys
+		Sum of bytes in variable-length index keys
		= Subtotal

+		(Subtotal / 256) + 1 (Overhead)
+		Number of variable-length columns + 1
+	2	(Overhead)
		= Clustered index row size

The results of the division (Subtotal / 256) are rounded down.

Compute the number of clustered index pages

Formula			Example		
(2016 / Clustered row size) - 2	=	No. of clustered index rows per page	(2016 / 9) - 2	=	222
No. of rows / No. of CI rows per page	=	No. of index pages at next level	750,000 / 222	=	3379

If the result for the “number of index pages at the next level” is greater than 1, repeat the following division step, using the quotient as the next dividend, until the quotient equals 1, which means that you have reached the root level of the index:

Formula

No. of index pages at last level	/	No. of clustered index rows per page	=	No. of index pages at next level

Example

3379 / 222	=	16 index pages (Level 1)
16 / 222	=	1 index page (Level 2)

Compute the total number of index pages

Add the number of pages at each level to determine the total number of pages in the index:

Formula			Example	
Index levels	Pages		Pages	Rows
2			1	16
1	+		+ 16	3379
0	+		+ 3379	750000
		Total number of index pages	3396	

Calculate allocation overhead and total pages

Each table and each index on a table has an object allocation map (OAM). A single OAM page holds allocation mapping for between 2,000 and 63,750 data pages or index pages. In most cases, the number of OAM pages required is close to the minimum value. To calculate the number of OAM pages for the table, use:

Formula			Example		
Number of reserved data pages / 63,750	=	Minimum OAM pages	750,000 / 63,750	=	12
Number of reserved data pages / 2000	=	Maximum OAM pages	750,000 / 2000	=	376

To calculate the number of OAM pages for the index, use:

Formula			Example		
Number of reserved index pages / 63,750	=	Minimum OAM pages	3396/ 63,750	=	1
Number of reserved index pages / 2000	=	Maximum OAM pages	3396 / 2000	=	2

Total pages needed

Finally, add the number of OAM pages to the earlier totals to determine the total number of pages required:

Formula			Example	
	Minimum	Maximum	Minimum	Maximum
Clustered index pages			3396	3396
OAM pages	+	+	1	2
Data pages	+	+	750000	750000
OAM pages	+	+	12	376
Total			753409	753773

Calculate the size of the leaf index row

Index rows containing variable-length columns require more overhead than index rows containing only fixed-length values.

Fixed-length keys only

If the index contains only fixed-length keys and are defined as NOT NULL, use:

Formula	
7	(Overhead)
+	Sum of fixed-length keys
	= Size of leaf index row

Some variable-length keys

If the index contains any variable-length keys or columns defined as NULL, use:

Formula		Example	
9	(Overhead)		9
+	Sum of length of fixed-length keys	+	4
+	Sum of length of variable-length keys	+	20
+	Number of variable-length keys + 1	+	2
	= Subtotal		35
+	(Subtotal / 256) + 1 (overhead)	+	1
	= Size of leaf index row		36

Calculate the number of leaf pages in the index

Formula			Example		
(2016 / leaf row size)	=	No. of leaf index rows per page	2016 / 36	=	56
No. of table rows / No. of leaf rows per page	=	No. of index pages at next level	9,000,000 / 56	=	160,715

Calculate the size of the nonleaf rows

Formula		Example
	Size of leaf index row	36
+ 4	Overhead	+ 4
	= Size of non-leaf row	40

Calculate the number of non-leaf pages

Formula			Example
(2016 / Size of non-leaf row) - 2	=	No. of non-leaf index rows per page	(2016 / 40) - 2 = 48

If the number of leaf pages from step 8 is greater than 1, repeat the following division step, using the quotient as the next dividend, until the quotient equals 1, which means that you have reached the root level of the index:

Formula

No. of index pages at previous level / No. of non-leaf index rows per page = No. of index pages at next level

Example

160715 / 48 = 3349	Index pages, level 1
3349 / 48 = 70	Index pages, level 2
70 / 48 = 2	Index pages, level 3
2 / 48 = 1	Index page, level 4 (root level)

Calculate the total number of non-leaf index pages

Add the number of pages at each level to determine the total number of pages in the index:

Index Levels	Pages		Pages	Rows
4			1	2
3	+	+	2	70
2	+	+	70	3348
1	+	+	3349	160715
0	+	+	160715	9000000
		Total number of 2K data pages used	164137	

Calculate allocation overhead and total pages

Formula			Example		
Number of index pages / 63,750	=	Minimum OAM pages	164137 / 63,750	=	3
Number of index pages / 2000	=	Maximum OAM pages	164137 / 2000	=	83

Total Pages Needed

Add the number of OAM pages to the total in step 11 to determine the total number of index pages:

Formula			Example	
	Minimum	Maximum	Minimum	Maximum
Nonclustered index pages			164137	164137
OAM pages	+	+	3	83
Total			164140	164220

Calculating the sizes of data-only-locked tables

The formulas and examples that follow show how to calculate the sizes of tables and indexes. This example uses the same column sizes and index as the previous example. All of the formulas use the maximum size of the variable-length fields. See "Tables and indexes used in the formulas" on page 89 for the specifications.

The formulas for data-only-locked tables are divided into two sets of steps:

- Steps 1–3 outline the calculations for a data-only-locked table. The example that follows step 3 illustrates the computations on a table that has 9,000,000 rows.
- Steps 4–8 outline the calculations for computing the space required by an index, followed by an example using the 9,000,000-row table.

optdiag output includes the average length of data rows and index rows. You can use these values for the data and index row lengths, if you want to use average lengths instead.

The steps are:

1 "Calculate the data row size" on page 97
2 "Compute the number of data pages" on page 98
3 "Calculate allocation overhead and total pages" on page 98
4 "Calculate the size of the index row" on page 98

5 "Calculate the number of leaf pages in the index" on page 99

6 "Calculate the number of non-leaf pages in the index" on page 99

7 "Calculate the total number of non-leaf index pages" on page 100

8 "Calculate allocation overhead and total pages" on page 100

Calculate the data row size

Rows that store variable-length data require more overhead than rows that contain only fixed-length data, so there are two separate formulas for computing the size of a data row.

Fixed-length columns only

If the table contains only fixed-length columns defined as NOT NULL, use:

6	(Overhead)
+	Sum of bytes in all fixed-length columns
	Data row size

Note Data-only-locked tables must allow room for each row to store a 6-byte forwarded row ID. If a data-only-locked table has rows shorter than 10 bytes, each row is padded to 10 bytes when it is inserted. This affects only data pages, and not indexes, and does not affect allpages-locked tables.

Some variable-length columns

If the table contains variable-length columns or columns that allow NULL values, use:

Formula		Example
8	(Overhead)	8
+	Sum of bytes in all fixed-length columns	+ 100
+	Sum of bytes in all variable-length columns	+ 50
+	Number of variable-length columns * 2	+ 4
	Data row size	162

Compute the number of data pages

Formula

2002 / Data row size = Number of data rows per page

Number of rows / Rows per page = Number of data pages required

In the first part of this step, the number of rows per page is rounded down:

Example

2002 / 162	=	12 data rows per page
9,000,000 / 12	=	750,000 data pages

Calculate allocation overhead and total pages

Allocation overhead

Each table and each index on a table has an object allocation map (OAM). The OAM is stored on pages allocated to the table or index. A single OAM page holds allocation mapping for between 2,000 and 63,750 data pages or index pages. In most cases, the number of OAM pages required is close to the minimum value. To calculate the number of OAM pages for the table, use:

Formula			Example		
Number of reserved data pages / 63,750	=	Minimum OAM pages	750,000 / 63,750	=	12
Number of reserved data pages / 2000	=	Maximum OAM pages	750,000 / 2000	=	375

Total pages needed

Add the number of OAM pages to the earlier totals to determine the total number of pages required:

Formula			Example	
	Minimum	Maximum	Minimum	Maximum
Data pages	+	+	750000	750000
OAM pages	+	+	12	375
Total			750012	750375

Calculate the size of the index row

Use these formulas for clustered and nonclustered indexes on data-only-length tables.

Index rows containing variable-length columns require more overhead than index rows containing only fixed-length values.

Fixed-length keys only

If the index contains only fixed-length keys defined as NOT NULL, use:

9	(Overhead)
+	Sum of fixed-length keys
	Size of index row

Some variable-length keys

If the index contains any variable-length keys or columns that allow NULL values, use:

Formula		Example	
9	(Overhead)		9
+	Sum of length of fixed-length keys	+	4
+	Sum of length of variable-length keys	+	20
+	Number of variable-length keys * 2	+	2
	Size of index row		35

Calculate the number of leaf pages in the index

Formula

2002 / Size of index row = No. of rows per page

No. of rows in table / No. of rows per page = No. of leaf pages

Example

2002 / 35 = 57 Nonclustered index rows per page

9,000,000 / 57 = 157,895 leaf pages

Calculate the number of non-leaf pages in the index

Formula

No. of leaf pages / No. of index rows per page = No. of pages at next level

If the number of index pages at the next level above is greater than 1, repeat the following division step, using the quotient as the next dividend, until the quotient equals 1, which means that you have reached the root level of the index:

Formula

No. of index pages at previous level / No. of non-leaf index rows per page = No. of index pages at next level

Example

157895/57 = 2771	Index pages, level 1
2770 / 57 = 49	Index pages, level 2
48 / 57 =1	Index pages, level 3

Calculate the total number of non-leaf index pages

Add the number of pages at each level to determine the total number of pages in the index:

Formula			**Example**	
Index levels	Pages		Pages	Rows
3			1	49
2	+		49	2771
1	+		2771	157895
0	+		157895	9000000
		Total number of 2K pages used	160716	

Calculate allocation overhead and total pages

Formula

Number of index pages / 63,750 = Minimum OAM pages

Number of index pages / 2000 = Maximum OAM pages

Example

160713 / 63,750 = 3 (minimum)

160713 / 2000 = 81 (maximum)

Total pages needed

Add the number of OAM pages to the total in step 8 to determine the total number of index pages:

Formula			Example	
	Minimum	Maximum	Minimum	Maximum
Nonclustered index pages			160716	160716
OAM pages	+	+	3	81
Total			160719	160797

Other factors affecting object size

In addition to the effects of data modifications that occur over time, other factors can affect object size and size estimates:

- Space management properties
- Whether computations used average row size or maximum row size
- Very small text rows
- Use of text and image data

Effects of space management properties

Values for fillfactor, exp_row_size, reservepagegap and max_rows_per_page can affect object size.

fillfactor

The fillfactor you specify for create index is applied when the index is created. The fillfactor is not maintained during inserts to the table. If a fillfactor has been stored for an index using sp_chgattribute, this value is used when indexes are re-created with alter table commands and reorg rebuild. The main function of fillfactor is to allow space on the index pages, to reduce page splits. Very small fillfactor values can cause the storage space required for a table or an index to be significantly greater.

See "Reducing index maintenance" on page 55 for details about setting fillfactor values.

exp_row_size

Setting an expected row size for a table can increase the amount of storage required. If your tables have many rows that are shorter than the expected row size, setting this value and running reorg rebuild or changing the locking scheme increases the storage space required for the table. However, the space usage for tables that formerly used max_rows_per_page should remain approximately the same.

See "Reducing row forwarding" on page 62 for details about setting exp_row_size values.

reservepagegap

Setting a reservepagegap for a table or an index leaves empty pages on extents that are allocated to the object when commands that perform extent allocation are executed. Setting reservepagegap to a low value increases the number of empty pages and spreads the data across more extents, so the additional space required is greatest immediately after a command such as create index or reorg rebuild. Row forwarding and inserts into the table fill in the reserved pages.

See "Leaving space for forwarded rows and inserts" on page 68.

max_rows_per_page

The max_rows_per_page value (specified by create index, create table, alter table, or sp_chgattribute) limits the number of rows on a data page.

To compute the correct values when using max_rows_per_page, use the max_rows_per_page value or the computed number of data rows per page, whichever is smaller, in "Compute the number of data pages" on page 91 and "Calculate the number of leaf pages in the index" on page 94.

See "Using max_rows_per_page on allpages-locked tables" on page 75.

Very small rows

For all-pages locked tables, Adaptive Server cannot store more than 256 data or index rows on a page. Even if your rows are extremely short, the minimum number of data pages is:

Number of Rows / 256 =	Number of data pages required

LOB pages

Each text or image or Java off-row column stores a 16-byte pointer in the data row with the datatype varbinary(16). Each column that is initialized requires at least 2K (one data page) of storage space.

Columns store implicit NULL values, meaning that the text pointer in the data row remains NULL and no text page is initialized for the value, saving 2K of storage space.

If a LOB column is defined to allow NULL values, and the row is created with an insert statement that includes NULL for the column, the column is not initialized, and the storage is not allocated.

If a LOB column is changed in any way with update, then the text page is allocated. Inserts or updates that place actual data in a column initialize the page. If the column is subsequently set to NULL, a single page remains allocated.

Each LOB page stores approximately 1800 bytes of data. To estimate the number of pages that a particular entry will use, use this formula:

Data length / 1800 = Number of 2K pages

The result should be rounded up in all cases; that is, a data length of 1801 bytes requires two 2K pages.

The total space required for the data may be slightly larger than the calculated value, because some LOB pages store pointer information for other page chains in the column. Adaptive Server uses this pointer information to perform random access and prefetch data when accessing LOB columns. The additional space required to store pointer information depends on the total size and type of the data stored in the column. Use Table 4-3 to estimate the additional pages required to store pointer information for data in LOB columns.

Table 4-3: Estimated additional pages for pointer information in LOB columns

Data size and type	Additional pages required for pointer information
400K image	0 to 1 page
700K image	0 to 2 pages
5MB image	1 to 11 pages
400K of multibyte text	1 to 2 pages
700K of multibyte text	1 to 3 pages
5MB of multibyte text	2 to 22 pages

Advantages of using formulas to estimate object size

The advantages of using the formulas are:

- You learn more details of the internals of data and index storage.
- The formulas provide flexibility for specifying averages sizes for character or binary columns.
- While computing the index size, you see how many levels each index has, which helps estimate performance.

Disadvantages of using formulas to estimate object size

The disadvantages of using the formulas are:

- The estimates are only as good as your estimates of average size for variable-length columns.
- The multistep calculations are complex, and skipping steps may lead to errors.
- The actual size of an object may be different from the calculations, based on use.

CHAPTER 5 Database Maintenance

This chapter explains how maintenance activities can affect the performance of other Adaptive Server activities, and how to improve the performance of maintenance tasks.

Topic	Page
Running reorg on tables and indexes	105
Creating and maintaining indexes	106
Creating or altering a database	110
Backup and recovery	112
Bulk-copy	113
Database consistency checker	117
Using dbcc tune (cleanup)	117
Using dbcc tune on spinlocks	117
Determining the space available for maintenance activities	118

Maintenance activities include tasks such as dropping and recreating indexes, performing dbcc checks, and updating table and index statistics. All of these activities can compete with other processing work on the server.

Whenever possible, perform maintenance tasks when your Adaptive Server usage is low. This chapter can help you determine the impact these activities have on individual applicaton performance, and on overall Adaptive Server performance.

Running *reorg* on tables and indexes

The reorg command can improve performance for data-only-locked tables by improving the space utilization for tables and indexes. The reorg subcommands and their uses are:

- reclaim_space – clears committed deletes and the space that is left when updates shorten the length of data rows.

- forwarded_rows – returns forwarded rows to home pages.
- compact – performs both of the operations above.
- rebuild – rebuilds an entire table or index. You can use reorg rebuild on both all-pages and data-only locked tables.

When you run reorg rebuild on a table, and the table is locked for the entire time it takes to rebuild the table and its indexes. Schedule the reorg rebuild command on a table when users do not need access to the table.

All of the other reorg commands, including reorg rebuild on an index, lock a small number of pages at a time, and use short, independent transactions to perform their work. You can run these commands at any time. The only negative effect might be on systems that are very I/O bound.

For more information on running reorg commands, see Chapter 9, “Using the reorg Command” in *System Administration Guide: Volume 2.*

Creating and maintaining indexes

When a user creates an index, all other users are locked out of the table.. The type of lock depends on the type of index:

- Creating a clustered index requires an exclusive table lock, locking out all table activity. Since rows in a clustered index are arranged in order by the index key, create clustered index reorders data pages.
- Creating a nonclustered index requires a shared table lock, locking out update activity.

Configuring Adaptive Server to speed sorting

Use the number of sort buffers configuration parameter to set the number of buffers that can be used in cache to hold pages from the input tables. In addition, parallel sorting can benefit from large I/O in the cache used to perform the sort.

See Chapter 5, “Parallel Query Processing” in *Performance and Tuning Series: Query Processing and Abstract Plans*.

Dumping the database after creating an index

When you create an index, Adaptive Server writes the create index transaction and the page allocations to the transaction log, but does not log the actual changes to the data and index pages. To recover a database that you have not dumped since you created the index, the entire create index process is executed again while loading transaction log dumps.

If you routinely re-create indexes (for example, to maintain the fillfactor in the index), you may want to schedule these operations to run shortly before a routine database dump.

Creating an index on sorted data

To recreate a clustered index, or to create one on data that was bulk copied into the server in index key order, use the sorted_data option to create index to shorten index creation time.

Since the data rows must be arranged in key order for clustered indexes, creating a clustered index without sorted_data requires you to rewrite the data rows to a complete new set of data pages. In some cases , Adaptive Server can skip sorting and copying the table's data rows: Factors include table partitioning and on clauses used in the create index statement.

When you are creating an index on a nonpartitioned table, sorted_data and the use of any of the following clauses requires you to copy the data, but does not require a sort:

- ignore_dup_row
- fillfactor
- The on *segment_name* clause, specifying a different segment from the segment where the table data is located
- The max_rows_per_page clause, specifying a value that is different from the value associated with the table

When these options and sorted_data are included in a create index on a partitioned table, the sort step is performed and the data is copied, distributing the data pages evenly on the table's partitions.

Table 5-1: Using options for creating a clustered index

Options	Partitioned table	Unpartitioned table
No options specified	Parallel sort; copies data, distributing evenly on partitions; creates index tree.	Either parallel or nonparallel sort; copies data, creates index tree.

Options	Partitioned table	Unpartitioned table
with sorted_data only or with sorted_data on *same_segment*	Creates index tree only. Does not perform the sort or copy data. Does not run in parallel.	Creates index tree only. Does not perform the sort or copy data. Does not run in parallel.
with sorted_data and ignore_dup_row or fillfactor or on *other_segment* or max_rows_per_page	Parallel sort; copies data, distributing evenly on partitions; creates index tree.	Copies data and creates the index tree. Does not perform the sort. Does not run in parallel.

In the simplest case, using sorted_data and no other options on a nonpartitioned table, the order of the table rows is checked and the index tree is built during this single scan.

If the data rows must be copied, but no sort needs to be performed, a single table scan checks the order of rows, builds the index tree, and copies the data pages to the new location in a single table scan.

For large tables that require numerous passes to build the index, saving the sort time considerably reduces I/O and CPU utilization.

When you create a clustered index that copies the data rows, the space available must be approximately 120 percent of the table size to copy the data and store the index pages.

Maintaining index and column statistics

The histogram and density values for an index are not maintained as data rows are added and deleted. The database owner must issue an update statistics command to ensure that statistics are current. Run update statistics after:

- Deleting or inserting rows that change the skew of key values in the index.
- Adding rows to a table for which rows were previously deleted with truncate table.
- Updating values in index columns.
- Inserts to any index that includes an IDENTITY column or any increasing key value. Date columns often have regularly increasing keys.

Running update statistics on these types of indexes is especially important if the IDENTITY column or other increasing key is the leading column in the index. After a number of rows have been inserted past the last key in the table when the index was created, all that the optimizer can tell is that the search value lies beyond the last row in the distribution page. It cannot accurately determine how many rows match a given value.

Note Failure to update statistics can severely impair performance.

See *Performance and Tuning Series: Improving Performance with Statistical Analysis*.

Rebuilding indexes

Rebuilding indexes reclaims space in the binary trees (a tree where all leaf pages are the same distance from the root page of the index). As pages are split and rows are deleted, indexes may contain many pages that contain only a few rows. Also, if the application performs scans on covering nonclustered indexes and large I/O, rebuilding the nonclustered index maintains the effectiveness of large I/O by reducing fragmentation.

You can rebuild indexes by dropping and recreating the index.

Rebuild indexes when:

- Data and usage patterns have changed significantly.
- A period of heavy inserts is expected, or has just been completed.
- The sort order has changed.
- Queries that use large I/O require more disk reads than expected, or optdiag reports lower cluster ratios than usual.
- Space usage exceeds estimates because heavy data modification has left many data and index pages partially full.
- Space for expansion provided by the space management properties (fillfactor, expected row size, and reserve page gap) has been filled by inserts and updates, resulting in page splits, forwarded rows, and fragmentation.
- dbcc has identified errors in the index.

If you recreate a clustered index or run reorg rebuild on a data-only-locked or all-pages-locked table, all nonclustered indexes are recreated, since creating the clustered index moves rows to different pages.

When system activity is low:

- Delete all indexes to allow more efficient bulk inserts.
- Create a new group of indexes to help generate a set of reports.

Creating or altering a database

Creating or altering a database is I/O-intensive; consequently, other I/O-intensive operations may suffer. When you create a database, Adaptive Server copies the model database to the new database and then initializes all the allocation pages and clears database pages.

To speed database creation or minimize its impact on other processes:

- Use the create database...for load option if you are restoring a database; that is, if you are getting ready to issue a load database command.

  When you create a database without for load, Adaptive Server copies model and then initializes all of the allocation units.

  When you use for load, Adaptive Server does initialize the allocation units until the load is complete. Then it initializes only the untouched allocation units. If you are loading a very large database dump, this can save a lot of time.

- Create databases during off-peak hours if possible.

create database and alter database perform concurrent, parallel I/O when clearing database pages. The number of devices is limited by the number of large i/o buffers configuration parameter. The default value for this parameter is 6, allowing parallel I/O on 6 devices at once.

A single create database and alter database command can use up to 32 of these buffers at once. These buffers are also used by load database, disk mirroring, and some dbcc commands.

Using the default value of 6, if you specify more than 6 devices, the first 6 writes are immediately started. As the I/O to each device completes, the 16K buffers are used for remaining devices listed in the command. The following example names 10 separate devices:

```
create database hugedb
      on dev1 = 100,
      dev2 = 100,
      dev3 = 100,
      dev4 = 100,
      dev5 = 100,
      dev6 = 100,
      dev7 = 100,
      dev8 = 100
log on logdev1 = 100,
      logdev2 = 100
```

During operations that use these buffers, a message is sent to the log when the number of buffers is exceeded. This information, for the create database command above, shows that create database started clearing devices on the first 6 disks, using all of the large I/O buffers, and then waited for them to complete before clearing the pages on other devices:

```
CREATE DATABASE: allocating 51200 pages on disk 'dev1'
CREATE DATABASE: allocating 51200 pages on disk 'dev2'
CREATE DATABASE: allocating 51200 pages on disk 'dev3'
CREATE DATABASE: allocating 51200 pages on disk 'dev4'
CREATE DATABASE: allocating 51200 pages on disk 'dev5'
CREATE DATABASE: allocating 51200 pages on disk 'dev6'
01:00000:00013:1999/07/26 15:36:17.54 server  No disk i/o buffers
are available for this operation. The total number of buffers is
controlled by the configuration parameter 'number of large i/o
buffers'.
CREATE DATABASE: allocating 51200 pages on disk 'dev7'
CREATE DATABASE: allocating 51200 pages on disk 'dev8'
CREATE DATABASE: allocating 51200  pages on disk 'logdev1'
CREATE DATABASE: allocating 51200  pages on disk 'logdev2'
```

Note In Adaptive Server version 12.5.0.3 and later, the size of the large I/O buffers used by create database, alter database, load database, and dbcc checkalloc is one allocation (256 pages), not one extent (8 pages), as it was in earlier versions. The server thus requires more memory allocation for large buffers. For example, a disk buffer that required memory for 8 pages in earlier versions now requires memory for 256 pages.

Backup and recovery

All Adaptive Server backups are performed by Backup Server. The backup architecture uses a client/server paradigm, with Adaptive Servers as clients to Backup Server.

Local backups

Adaptive Server sends the local Backup Server instructions, via remote procedure calls, telling the Backup Server which pages to dump or load, which backup devices to use, and other options. Backup Server performs all the disk I/O.

Adaptive Server does not read or send dump and load data, it sends only instructions.

Remote backups

Backup Server also supports backups to remote machines. For remote dumps and loads, a local Backup Server performs the disk I/O related to the database device and sends the data over the network to the remote Backup Server, which stores it on the dump device.

Online backups

You can perform backups while a database is active. Clearly, such processing affects other transactions, but you should not hesitate to back up critical databases as often as necessary to satisfy the reliability requirements of the system.

See the *System Administration Guide, Volume 2* for a complete discussion of backup and recovery strategies.

Using thresholds to prevent running out of log space

If your database has limited log space, and you occasionally hit the *last-chance threshold*, install a second threshold that provides ample time to perform a transaction log dump. Running out of log space has severe performance impacts. Users cannot execute any data modification commands until log space has been freed.

Minimizing recovery time

You can help minimize recovery time by changing the recovery interval configuration parameter. The default value of 5 minutes per database works for most installations. Reduce this value only if functional requirements dictate a faster recovery period. Reducing the value increases the amount of I/O required.

See Chapter 5, "Memory Use and Performance," in *Performance and Tuning Series: Basics*.

Recovery speed may also be affected by the value of the housekeeper free write percent configuration parameter. The default value of this parameter allows the server's housekeeper wash task to write dirty buffers to disk during the server's idle cycles, as long as disk I/O is not increased by more than 20 percent.

Recovery order

During recovery, system databases are recovered first. Then, user databases are recovered in order by database ID.

Bulk-copy

Bulk-copying into a table on Adaptive Server runs fastest when there are no clustered indexes on the table and you have enabled select into/ bulkcopy. If you have not enabled this option, slow bcp is used for tables with any index or active trigger.

fast bcp logs page allocation only for tables without an index. fast bcp saves time because it does not update indexes for each data insert, nor does it log the changes to the index pages. However, if you use fast bcp on a table with an index, it does log index updates.

fast bcp is automatically used for tables with triggers. To use slow bcp, disable the select into/bulk copy database option while you perform the copy.

To use fast bulk-copy:

1. Use sp_dboption to set the select into/bulkcopy/pllsort option. Remember to disable the option after the bulk-copy operation completes.
2. Drop any clustered indexes. Recreate them when the bulk-copy completes.

Note You need not deactivate triggers during the copy.

During fast bulk-copy, rules are not enforced, but defaults are.

Since changes to the data are not logged, perform a dump database soon after a fast bulk-copy operation. Performing a fast bulk-copy in a database blocks the use of dump transaction, since the unlogged data changes cannot be recovered from the transaction log dump.

Parallel bulk-copy

For fastest performance, use fast bulk-copy to copy data into partitioned tables. For each bulk-copy session, specify the partition on which the data should reside.

If your input file is already in sorted order, you can bulk-copy data into partitions in order, and avoid the sorting step while creating clustered indexes.

See Chapter 10, “Partitioning Tables and Indexes,” in the *Transact-SQL Users Guide* for step-by-step procedures.

Batches and bulk-copy

If you specify a batch size during a fast bulk-copy, each new batch must start on a new data page, since only the page allocations, and not the data changes, are logged during a fast bulk-copy. Copying 1000 rows with a batch size of 1 requires 1000 data pages and 1000 allocation records in the transaction log.

If you use a small batch size to help detect errors in the input file, you may want to choose a batch size that corresponds to the numbers of rows that fit on a data page.

Slow bulk-copy

By default, Adaptive Server uses slow bcp by default if a table has a clustered index, index, or trigger with the select into/bulk copy enabled.

For slow bulk-copy:

- You do not have to set select into/bulkcopy.
- Rules are not enforced and triggers are not fired, but defaults are enforced.
- All data changes are logged, as are page allocations.
- Indexes are updated as rows are copied in, and index changes are logged.

Improving bulk-copy performance

Other ways to increase bulk-copy performance are:

- Set the trunc log on chkpt option to keep the transaction log from filling up. If your database has a threshold procedure that automatically dumps the log when it fills, you save the transaction dump time

  Each batch is a separate transaction, so if you do not specify a batch size, setting trunc log on chkpt does not improve performance.

- Set the number of pre-allocated extents configuration parameter high if you perform many large bulk copies.

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

- Find the optimal network packet size.

  See Chapter 2, “Networks and Performance,” in *Performance and Tuning Series: Basics*.

Replacing the data in a large table

If you are replacing all the data in a large table, use truncate table, which performs reduced logging, instead of delete. Only page deallocations are logged.

1. Truncate the table.
2. Drop all indexes on the table.
3. Load the data.
4. Recreate the indexes.

See the *Reference Manual: Commands*.

Adding large amounts of data to a table

When you are adding 10 – 20 percent or more to a large table, drop the nonclustered indexes, load the data, and then recreate nonclustered indexes.

For very large tables, you may need to leave the clustered index in place due to space constraints. Adaptive Server must make a copy of the table when it creates a clustered index. In many cases, once tables become very large, the time required to perform a slow bulk-copy with the index in place is less than the amount of time it takes to perform a fast bulk-copy and recreate the clustered index.

Using partitions and multiple bulk-copy processes

If you load data into a table without indexes, you can create partitions on the table and use one bcp session for each partition.

See Chapter 4, “Using bcp to Transfer Data to and from Adaptive Server” in the *Utility Guide*.

Impacts on other users

Bulk-copying large tables in or out may affect response time for other users. If possible:

- Schedule bulk-copy operations for off-peak hours.

- Use fast bulk-copy, since it performs less logging and less I/O.

Database consistency checker

Periodically, use dbcc to run database consistency checks. If you back up a corrupt database, the backup is useless. dbcc affects performance, since dbcc must acquire locks on the objects it checks.

See Chapter 10, "Checking Database Consistency" in the *System Administration Guide: Volume 2* for information about dbcc and locking, with additional information about how to minimize the effects of dbcc on user applications.

Using *dbcc tune (cleanup)*

Adaptive Server performs redundant memory cleanup checking as a final integrity check after processing each task. In very high throughput environments, you may realize a slight performance improvement by skipping this cleanup error check. To turn off error checking, enter:

```
dbcc tune(cleanup,1)
```

The final cleanup frees any memory a task might hold. If you turn error checking off, but you get memory errors, reenable the checking by entering:

```
dbcc tune(cleanup,0)
```

Using *dbcc tune* on spinlocks

"When you see a scaling problem resulting from spinlock contention, use des_bind to improve the scalability of the server where object descriptors are reserved for hot objects. Descriptors for bound objects are never released. Binding the descriptors for even a few commonly used objects may reduce the overall metadata spinlock contention and improve performance.

```
dbcc tune(des_bind, <dbid>, <objname>)
```

To remove the binding, use:

```
dbcc tune(des_unbind, <dbid>, <objname>)
```

Note To unbind an object from the database, the database must be in single user mode.

Do not use des_bind:

- On objects in system databases such as master and tempdb
- On system tables

Since des_bind is not persistent, you must reissue any binding commands each time you restart the server.

Determining the space available for maintenance activities

Several maintenance operations require room to make a copy of the data pages of a table:

- create clustered index
- alter table...lock
- Some alter table commands that add or modify columns
- alter table...partition by
- reorg rebuild on a table

In most cases, these commands also require space to recreate any indexes, so you must determine:

- The size of the table and its indexes
- The amount of space available on the segment where the table is stored
- The space management properties set for the table and its indexes

Overview of space requirements

Any command that copies a table's rows also recreates all of the indexes on the table.You need enough available space for a complete copy of the table and copies of all indexes.

These commands do not estimate how much space is needed. If a command runs out of space on any segment used by the table or its indexes the command stops, and issues an error message. For large tables, can occur minutes, even hours, after the command starts.

You need free space on the segments used by the table and its indexes, as follows:

- Free space on the table's segment must be at least equal to:
    - The size of the table, plus
    - Approximately 20 percent of the table size, if the table has a clustered index and you are changing from allpages locking to data-only locking.
- Free space on the segments used by nonclustered indexes must be at least equal to the size of the indexes.

Clustered indexes for data-only-locked tables have a leaf level above the data pages. If you alter a table with a clustered index from allpages locking to data-only locking, the resulting clustered index requires more space. The additional space required depends on the size of the index keys.

Checking space usage and space available

As a simple guideline, copying a table and its indexes requires space equal to the current space used by the table and its indexes, plus about 20% additional space. However:

- If data modifications have created many partially full pages, the space requirement for the copy of the table can be smaller than the current size.
- If space-management properties for the table have changed, or if space required by fillfactor or reservepagegap has been filled by data modifications, the size required for the copy of the table can be larger.
- Adding columns or modifying columns to larger datatypes requires more space for the copy.

Log space is also required. Because Adaptive Server processes reorg rebuild as a single transaction, the amount of log space required can be large, particularly if the table it is rebuilding has multiple nonclustered indexes. Each nonclustered index requires log space, and there must be sufficient log space to create all indexes.

Checking space used for tables and indexes

To see the size of a table and its indexes, use:

```
sp_spaceused titles, 1
```

See "Calculating the sizes of data-only-locked tables" on page 96 for information on estimating the size of the clustered index.

Checking space on segments

Tables are always copied to free space on the segment where they are currently stored, and indexes are recreated on the segment where they are currently stored. Commands that create clustered indexes can specify a segment. The copy of the table and the clustered index are created on the target segment.

To determine the number of pages available on a segment, use sp_helpsegment. The last line of sp_helpsegment shows the total number of free pages available on a segment.

This command prints segment information for the default segment, where objects are stored when no segment was explicitly specified:

```
sp_helpsegment "default"
```

sp_helpsegment reports the names of indexes on the segment. If you do not know the segment name for a table, use sp_help and the table name. The segment names for indexes are also reported by sp_help.

Checking space requirements for space management properties

If you make significant changes to space management property values, the table copy can be considerably larger or smaller than the original table. Settings for space management properties are stored in the sysindexes tables, and are displayed by sp_help and sp_helpindex. This output shows the space management properties for the titles table:

```
exp_row_size reservepagegap fillfactor max_rows_per_page
------------ -------------- ---------- -----------------
         190             16         90                 0
```

sp_helpindex produces this report:

```
index_name          index_description
    index_keys
    index_max_rows_per_page index_fillfactor index_reservepagegap
    ----------------------- ---------------- --------------------
title_id_ix         nonclustered located on default
    title_id
                         0               75                   0
title_ix            nonclustered located on default
    title
                         0               80                  16
type_price          nonclustered located on default
    type, price
                         0               90                   0
```

Space management properties applied to the table

During the copy step, the space management properties for the table are used as follows:

- If an expected row size value is specified for the table, and the locking scheme is being changed from allpages locking to data-only locking, the expected row size is applied to the data rows as they are copied.

  If no expected row size is set, but there is a max_rows_per_page value for the table, an expected row size is computed, and that value is used.

  Otherwise, the default value specified with the configuration parameter default exp_row_size percent is used for each page allocated for the table.
- The reservepagegap is applied as extents are allocated to the table.
- If sp_chgattribute has been used to save a fillfactor value for the table, it is applied to the new data pages as the rows are copied.

Space management properties applied to the index

When indexes are rebuilt, space management properties for the indexes are applied, as follows:

- If sp_chgattribute has been used to save fillfactor values for indexes, these values are applied when the indexes are recreated.
- If reservepagegap values are set for indexes, these values are applied when the indexes are recreated.

Estimating the effects of space management properties

Table 5-2 shows how to estimate the effects of setting space management properties.

Table 5-2: Effects of space management properties on space use

Property	Formula	Example
fillfactor	Requires (100/fillfactor) * *num_pages* if pages are currently fully packed	fillfactor of 75 requires 1.33 times current number of pages; a table of 1,000 pages grows to 1,333 pages.
reservepagegap	Increases space by 1/reservepagegap if extents are currently filled	reservepagegap of 10 increase space used by 10%; a table of 1,000 pages grows to 1,100 pages.
max_rows_per_page	Converted to exp_row_size when converting to data-only-locking	See Table 5-3 on page 122.
exp_row_size	Increase depends on number of rows smaller than exp_rowsize, and the average length of those rows	If exp_row_size is 100, and 1,000 rows have a length of 60, the increase in space is: (100 - 60) * 1000 or 40,000 bytes; approximately 20 additional pages.

See Chapter 3, "Setting Space Management Properties."

If a table has max_rows_per_page set, and the table is converted from allpages locking to data-only locking, the value is converted to an exp_row_size value before the alter table...lock command copies the table to its new location.

exp_row_size is enforced during the copy. Table 5-3 shows how the values are converted.

Table 5-3: Converting max_rows_per_page to exp_row_size

If *max_rows_per_page* is set to	Set *exp_row_size* to
0	Percentage value set by default exp_row_size percent
1 – 254	The smaller of: • Maximum row size • 2K logical page – 2002/max_rows_per_page value 4K logical page – 4050/max_rows_per_page value 8K logical page – 8146/max_rows_per_page value 16K logical page – 16338/max_rows_per_page value

If there is not enough space

If you do not have enough space to copy the table and recreate all the indexes, determine whether dropping the nonclustered indexes on the table leaves enough room to create a copy of the table. Without any nonclustered indexes, the copy operation requires space just for the table and the clustered index.

Do not drop the clustered index, since it is used to order the copied rows, and attempting to recreate it later may require space to make a copy of the table. Recreate the nonclustered indexes when the command completes.

CHAPTER 6 Temporary Databases

This chapter discusses performance issues associated with temporary databases. Temporary databases are server-wide resources, and are used primarily for processing sorts, creating worktables, reformatting, and storing temporary tables and indexes created by users. Anyone can create objects in temporary databases. Many processes use them silently.

Many applications use stored procedures that create tables in temporary databases to expedite complex joins or to perform other complex data analysis that cannot be performed easily in a single step.

Topic	Page
How temporary database management affects performance	125
Using temporary tables	126
Temporary databases	128
Session-assigned temporary database	128
Using multiple temporary databases	129
Tuning system temporary databases for performance	131
Logging optimizations for temporary databases	140

How temporary database management affects performance

Good management of temporary databases is critical to the overall performance of Adaptive Server. However, temporary tables can add to the size requirement of tempdb. Using temporary tables greatly affects the performance of Adaptive Server and your applications. You cannot overlook the management of temporary databases or leave them in a default state. On many servers, tempdb is the most dynamic database.

You can avoid most of the performance issues with temporary databases by planning in advance, and taking these issues into consideration:

- Temporary databases fill frequently, generating error messages to users, who must then resubmit their queries when space becomes available.
- Temporary databases sort slowly and queries against them display uneven performance.
- User queries are often temporarily blocked from creating temporary tables because of locks on system tables.
- Heavily used objects in a temporary database flush other pages out of the data cache.

Resolve these issues by:

- Configuring a sufficient number of user temporary databases.
- Sizing temporary databases correctly for all Adaptive Server activity
- Placing temporary databases optimally to minimize contention
- Minimizing the resource locking within temporary databases
- Binding temporary databases to their own data cache
- Configuring temporary database groups correctly
- Binding logins and applications to the appropriate temporary database or group.

Using temporary tables

Tables created in temporary database are called temporary tables. Use the temporary database to create different types of temporary tables. The types of temporary tables are:

- Hashed (#) temporary tables
- Regular user tables
- Worktables

Hashed (#) temporary tables

Hashed temporary tables:

- Exist only for the duration of the user session or for the scope of the procedure that creates them, and can be either manually or automatically dropped at the end of the session or procedure.
- Cannot be shared between user connections
- Are created in the temporary database assigned for the session.

Create hashed temporary tables by including a hash mark (“#”) as the first character of the table name:

```
create table #temptable (...)
```

or:

```
select select_list
into #temptable ...
```

When you create indexes on temporary tables, the indexes are stored in the same session assigned to the temporary database where the hashed table resides:

```
create index littletableix on #littletable(col1)
```

Regular user tables

To create regular user tables in a temporary table, specify the database name in the create table command:

```
create table tempdb..temptable (...)
```

Regular user tables in the temporary database:

- Can persist across sessions
- Can be used by bulk copy (bcp) operations
- Can be shared by granting permissions on them
- Must either be explicitly dropped by the owner or are automatically removed when Adaptive Server is restarted

or:

```
select select_list
into tempdb..temptable
```

You can create indexes on regular user tables created in the temporary database:

```
create index tempix on tempdb..temptable(col1)
```

Worktables

Adaptive Server creates internal temporary tables for the session-assigned tempdb for merges, sorts, joins, and so on. These temporary tables are called worktables, and they:

- Are never shared
- Disappear as soon as the command completes

Temporary databases

To avoid performance concerns that result from using a single temporary databases, you can create multiple temporary databases.

Adaptive Server includes one system-created temporary database called tempdb, which is created on the master device when you install Adaptive Server.

In addition to tempdb, Adaptive Server allows users to create multiple temporary databases. User-created temporary databases are similar to the system tempdb: they are used primarily to create temporary objects, and are recreated instead of recovered during start-up. Unlike tempdb, you can drop user-created temporary databases.

Multiple temporary databases:

- Reduce contention on system catalogs and log files in the system tempdb
- Can be created on fast access devices
- Can be created or dropped as needed.

Session-assigned temporary database

When a client connects, Adaptive Server assigns a temporary database to its session. Adaptive Server uses this session-assigned temporary database as a default space where it creates temporary objects (including hashed-temporary tables and worktables) for work the client performs. The session-assigned temporary database remains assigned to the session until the session connects to the client.

Adaptive Server selects temporary databases for a session according to these rules:

- If a binding already exists for a login, that binding is used.
- If an application name is specified and it has a binding, use that binding.
- If Adaptive Server does not find a binding, it assigns a temporary database from the default group using a round-robin scheme.

To specify that Adaptive Server creates an object in a specific temporary database. For example:

```
create procedure inv_amounts as
     select stor_id, "Total Due" = sum(amount)
     from #tempstores
     group by stor_id
```

Using multiple temporary databases

This section discussing how to create, configure, bind, and select temporary databases.

Creating user temporary databases

Create multiple temporary databases using the temporary database keyword in the create database syntax:

create temporary database *temporary_database_name* on
device_name=size log on *device_name*=size

For example, to create a user temporary database named tempdb_1 on the tempdb_device, enter:

```
create temporary database tempdb_1 on tempdb_device = 3
log on log_device = 1
```

Configuring a default tempdb group

Adaptive Server includes a group of temporary databases called the default group. When Adaptive Server starts a session, it selects a temporary database from the default group (using a round-robin technique) in which all temporary database activities are performed. Adaptive Server assigns this temporary database to the session. sp_who displays this temporary database in the tempdbname column. The round-robin scheme allows Adaptive Server to distribute the load evenly across all temporary databases in the default group because a single temporary database from the group is not performing all activities.

Initially, the default group consists only of tempdb. However, users may add multiple user databases to the default group. Use sp_tempdb to add a user database to the default group. For example, to add tempdb_1 to the default group, use:

```
sp_temodb "add'", "tempdb_1" , "default"
```

To drop tempdb_1 from the default group, use:

```
sp_tempdb "drop", "tempdb_1" , "default"
```

See the *Reference Manual: Procedures* for the complete sp_tempdb syntax.

Binding to groups and tempdb

The sp_tempdb. . . 'bind'...'unbind' system procedure allows you to bind, or unbind, an application or login to specific temporary database or tempdb group. After you create the binding, when the application or login connects to the server, Adaptive Server assigns the specified temporary database or temporary database group to which it is bound. Binding allows you to control the temporary database assignments for specific applications or logins.

This example binds the log in sa to the default group:

```
sp_tempdb 'bind', 'lg', 'sa', 'GR', 'default'
```

This example unbinds the login sa:

```
sp_tempdb 'unbind', 'lg', 'sa'
```

See Reference Manual: Procedures for the complete sp_tempdb syntax.

Binding applications and logins to temporary databases

Identify your application and login requirements for temporary databases. Bind these applications and logins to different databases or default groups to distribute the load evenly across available temporary databases to avoid catalog contention. Inappropriate bindings do not solve catalog contention even if there is a sufficient number of temporary databases—Adaptive Server may not distribute the load evenly across the temporary databases. See "Binding to groups and tempdb" on page 130.

Tuning system temporary databases for performance

This section discusses configuration issues related to temporary databases.

Placing system *tempdb*

When deciding where to place tempdb:

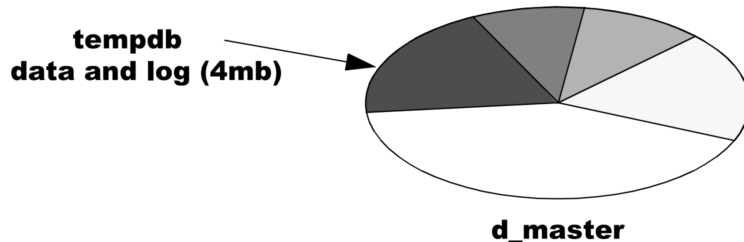
- Keep tempdb on separate physical disks than your critical application databases.
- Use the fastest disks available. If your platform supports solid state devices and tempdb use is a bottleneck for your applications, use those devices.
- After you expand tempdb onto additional devices, drop the master device from the system, default, and logsegment segments.

Although you can expand tempdb on the same device as the master database, Sybase suggests that you use separate devices. Also, remember that logical devices, but not databases, are mirrored using Adaptive Server mirroring. If you mirror the master device, you create a mirror of all portions of the databases that reside on the master device. If the mirror uses serial writes, this can have a serious performance impact if tempdb is heavily used.

Initial allocation of system *tempdb*

When you install Adaptive Server, the size of tempdb is 4MB, and is located completely on the master device, as shown in Figure 6-1. tempdb is typically the first database that a system administrator needs to make larger. The more users on the server, the larger it needs to be. Depending on your needs, you may want to stripe tempdb across several devices.

Figure 6-1: tempdb default allocation



Use sp_helpdb to see the size and status of tempdb. The following example shows tempdb defaults at installation time:

```
                       sp_helpdb tempdb
name       db_size  owner   dbid     created               status
--------- -------- ------ ------ -----------  --------------------
tempdb       2.0 MB      sa      2 May 22, 1999    select into/bulkcopy

device_frag      size        usage     free kbytes
------------ -------- ------------        --------
master          2.0 MB   data and log            1248
```

Dropping the master device from *tempdb* segments

By default, the system, default, and logsegment segments for tempdb include its 4MB allocation on the master device. When you allocate new devices to tempdb, they automatically become part of all three segments unless you add them as dedicated data or log. Once you allocate a second device to tempdb, you can drop the master device from the default, system, and logsegment segments. This way, you can be sure that the worktables and other temporary tables in tempdb do not contend with other uses on the master device.

To drop the master device from the segments:

1 Alter tempdb onto another device, if you have not already done so. For example:

```
alter database tempdb on tune3 = 20
```

2 Issue a use tempdb command, and then drop the master device from the segments:

```
sp_dropsegment "default", tempdb, master
sp_dropsegment "system", tempdb, master
sp_dropsegment "logsegment", tempdb, master
```

3 To verify the segments no longer include the master device, issue this command against the master database:

```
select dbid, name, segmap
from sysusages, sysdevices
where sysdevices.vdevno= sysusages.vdevno
and dbid = 2
and (status&2=2 or status&3=3))
```

The segmap column should report “0” for any allocations on the master device, indicating that no segment allocations exist:

```
 dbid   name            segmap
------ --------------- -----------
     2          master           0
     2           tune3           7
```

Alternatively, issue:

```
use tempdb
sp_helpdb 'tempdb'
device_fragments     size      usage             created   free kbytes
-----------------   ------   ----------  -----------------  ----------
master              4.0 MB   data only     Feb 7 2008 2:18AM      2376
tune3               20.0 MB  data and log May 16 2008 1:55PM     16212

device    segment
--------- ----------------------------
master    -- unused by any segments --
tune3                         default
tune3                      logsegment
tune3                          system
```

Configuring user-created temporary databases

Applications have individual resource and space requirements for temporary databases. Unless you understand your applications requirements, and maintain application to database or group bindings that satisfy these database requirements, make all temporary databases the same size. If all temporary databases are the same size, applications should not run out of resources or space, regardless of which database is assigned an application or session.

Caching user temporary databases

Generally, configure caches similarly across temporary databases within a group. The query processor may choose a query plan based on these caching characteristics, and you may see poor performance if the plan is executed using a cache with a different configuration.

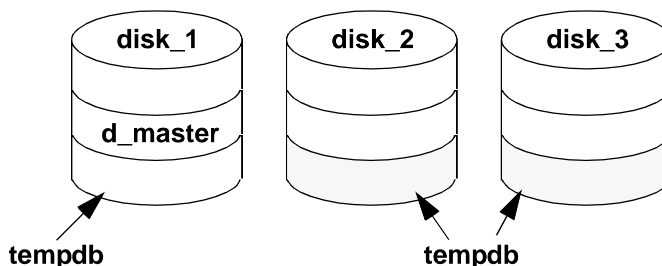
General guidelines

This section provide general guidelines for configuring the temporary databases, which apply to both system and user temporary databases.

Using multiple disks for parallel query performance

If temporary databases span multiple devices, as shown in Figure 6-2, you can take advantage of parallel query performance for some temporary tables or worktables.

Figure 6-2: tempdb spanning disks



Binding *tempdb* to its own cache

Under normal Adaptive Server use, temporary databases make heavy use of the data cache as temporary tables are created, populated, and dropped.

Assigning a temporary database to its own data cache:

- Keeps the activity on temporary objects from flushing other objects out of the default data cache
- Helps spread I/O between multiple caches

Commands for cache binding

Use sp_cacheconfig and sp_poolconfig to create named data caches and to configure pools of a given size for large I/O. Only a system administrator can configure caches and pools.

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 instructions on configuring named caches and pools, see Chapter 4, “Configuring Data Caches” in the *System Administration Guide: Volume 2*.

Once the caches have been configured, and the server has been restarted, you can bind tempdb to the new cache:

```
sp_bindcache "tempdb_cache", tempdb
```

Determining the size of temporary databases

Allocate sufficient space to temporary databases to handle the following processes for every concurrent Adaptive Server user:

- Worktables for merge joins
- Worktables that are created for distinct, group by, and order by, for reformatting, and for the or strategy, and for materializing some views and subqueries
- Hashed temporary tables (those created with "#" as the first character of their names)
- Indexes on temporary tables
- Regular user tables in temporary databases
- Procedures built by dynamic SQL

Some applications may perform better if you use temporary tables to split up multitable joins. This strategy is often used for:

- Cases where the optimizer does not choose a good query plan for a query that joins more than four tables
- Queries that join a very large number of tables
- Very complex queries
- Applications that need to filter data as an intermediate step

You might also use temporary databases to:

- Denormalize several tables into a few temporary tables
- Normalize a denormalized table to do aggregate processing

Determine the sizes of temporary databases based on usage scenarios. For most applications, make temporary databases 20 – 25% of the size of your user databases to provide enough space for these uses.

Minimizing logging in temporary databases

Even though the trunc log on checkpoint database option is turned on in temporary databases, Adaptive Server still writes changes to temporary databases to the transaction log. You can reduce log activity in a temporary database by:

- Using select into instead of create table and insert

- Selecting only the columns you need into the temporary tables

Using *select into*

When you create and populate temporary tables in a temporary database, use the select into command, rather than create table and insert...select, whenever possible. The select into/bulkcopy database option is turned on by default in temporary databases to enable this behavior.

select into operations are faster because they are only minimally logged. Only the allocation of data pages is tracked, not the actual changes for each data row. Each data insert in an insert...select query is fully logged, resulting in more overhead.

Using shorter rows

If the application creating tables in a temporary database uses only a few columns of a table, you can minimize the number and size of log records by:

- Selecting only the columns you need for the application, rather than using select * in queries that insert data into the tables
- Limiting the rows selected to just the rows that the applications requires

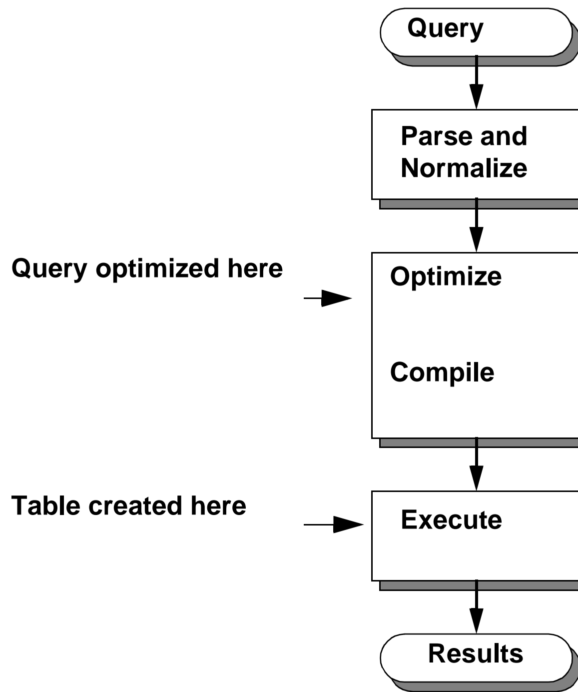
These suggestions also keep the size of the tables themselves smaller.

Optimizing temporary tables

Many uses of temporary tables are simple and brief and require little optimization. However, if your applications require multiple accesses to tables in a temporary database, examine them for possible optimization strategies. Usually, this involves splitting out the creation and indexing of the table from the access to it by using more than one procedure or batch.

When you create a table in the same stored procedure or batch where it is used, the query optimizer cannot determine how large the table is because the table was not created when the query was optimized, as shown in Figure 6-3. This applies to both temporary tables and regular user tables.

Figure 6-3: Optimizing and creating temporary tables



The optimizer assumes that any such table has 10 data pages and 100 rows. If the table is really large, this assumption can lead the optimizer to choose a suboptimal query plan.

These two techniques can improve the optimization of temporary tables:

- Creating indexes on temporary tables
- Breaking complex use of temporary tables into multiple batches or procedures to provide information for the optimizer

Creating indexes on temporary tables

You can define indexes on temporary tables. In many cases, these indexes can improve the performance of queries that use temporary databases. The optimizer uses these indexes just like indexes on ordinary user tables. The only requirements are:

- The table must contain data when the index is created. If you create the temporary table and create the index on an empty table, Adaptive Server does not create column statistics such as histograms and densities. If you insert data rows after creating the index, the optimizer has incomplete statistics.
- The index must exist while the query using it is optimized. You cannot create an index and then use it in a query in the same batch or procedure. The query processor uses indexes created in a stored procedure in queries that are run inside the stored procedure.
- The optimizer may choose a suboptimal plan if rows have been added or deleted since the index was created or since update statistics was run.

Providing an index for the optimizer can greatly increase performance, especially in complex procedures that create temporary tables and then perform numerous operations on them.

Creating nested procedures with temporary tables

You need to take an extra step to create the procedures described above. You cannot create base_proc until select_proc exists, and you cannot create select_proc until the temporary table exists.

1. Create the temporary table outside the procedure. It can be empty; it just must exist and have columns that are compatible with select_proc:

```
select * into #huge_result from ... where 1 = 2
```

2. Create the procedure select_proc, as shown above.
3. Drop #huge_result.
4. Create the procedure base_proc.

Breaking *tempdb* uses into multiple procedures

For example, this query causes optimization problems with #huge_result:

```
create proc base_proc
as
    select *
        into #huge_result
        from ...
    select *
        from tab,
        #huge_result where ...
```

You can achieve better performance by using two procedures. When the base_proc procedure calls the select_proc procedure, the optimizer can determine the size of the table:

```
create proc select_proc
as
    select *
        from tab, #huge_result where ...
create proc base_proc
as
    select *
        into #huge_result
        from ...
    exec select_proc
```

If the processing for #huge_result requires multiple accesses, joins, or other processes (such as looping with while), creating an index on #huge_result may improve performance. Create the index in base_proc so that it is available when select_proc is optimized.

Logging optimizations for temporary databases

Adaptive Server does not recover temporary databases when you shut it down or it fails, but Adaptive Server does create the temporary databases when you restart the server. Because temporary databases do not require recovery, Adaptive Server optimizes the logging mechanism for temporary databases to improve performance by:

- Single log records – force Adaptive Server to flush syslogs to disk immediately after Adaptive Server logs the record. Adaptive Server creates single log records while modifying OAM pages or allocation pages (in a database that is configured to use mixed log and data on the same device). Adaptive Server must flush syslogs to avoid undetected deadlocks created during buffer pinning. Because Adaptive Server does not pin buffers for temporary databases, it need not flush the syslogs data for the temporary database when it writes an single log records, which reduces log semaphore contention.

- Flushing dirty pages to disk – for databases that require recovery, Adaptive Server flushes dirty pages to disk during the checkpoint, ensuring that, if Adaptive Server fails, all committed data is saved to disk. For temporary databases, Adaptive Server supports runtime rollbacks, but not failure recovery, allowing it to avoid flushing dirty data pages at the checkpoint.
- Avoiding write-ahead logging – write-ahead logging guarantees that Adaptive Server can recover data for committed transactions by reissuing the transactions listed in the log, and undoing the changes performed by aborted or rolled back transactions. Adaptive Server does not support write-ahead logging on databases that do not require recovery. Because Adaptive Server does not recover temporary database, buffers for temporary databases are not pinned, which allows Adaptive Server to skip flushing the temporary database log when it commits a transaction using a temporary database.

User log cache (ULC)

Adaptive Server contains a separate user log cache (ULC) for the temporary database assigned to the session. The ULC allows Adaptive Server to avoid log flushes when users switch between a user database and session's temporary database, or if all the following conditions are met:

- Adaptive Server is currently committing the transaction.
- All the log records are in the ULC.
- There are no post-commit log records.

The configuration option, session tempdb log cache size, which allows you to configure the size of the ULC, helps determine how often it needs flushing. See Chapter 5, "Setting Configuration Parameters," in the *System Administration Guide: Volume 1*.

Index

A

access
    index 21
    optimizer methods 21
Adaptive Server
    logical page sizes 23
aggregate functions
    denormalization and temporary tables 136
allocation map. See object allocation map (OAM) pages
allocation pages 27
allocation units 25, 27
    database creation and 110
allpages-locked table, inserting data 41
alter table command
    lock option and **fillfactor** and 60
    reservepagegap for indexes 70
APL tables. See allpages locking
application design
    temporary tables in 136
auditing
    disk contention and 3

B

Backup Server 112
batch processing
    bulk copy and 114
    temporary tables and 138
bcp (bulk copy utility) 113
    heap tables and 41
    reclaiming space with 47
binding
    objects to data caches 49
    tempdb 135
buffers
    allocation and caching 52
    chain of 49
bulk copying. See bcp (bulk copy utility)

C

cache replacement strategy 50–53
caches, data
    aging in 49
    binding objects to 49
    data modification and 51
    deletes on heaps and 53
    I/O configuration 45
    inserts to heaps and 52
    joins and 50
    MRU replacement strategy 50
    pools in 45
    tempdb bound to own 135
    updates to heaps and 53
    wash marker 49
chain of buffers (data cache) 49
chains of pages
    placement 2
cluster ratio
    reservepagegap and 68, 73
clustered indexes
    computing number of data pages 98
    computing number of pages 91
    computing size of rows 91
    estimating size of 89, 96
    exp_row_size and row forwarding 62–67
    fillfactor effect on 58
    overhead 39
    performance and 39
    reclaiming space with 46
    reducing forwarded rows 62–67
    segments and 10
    size of 83, 92
columns
    datatype sizes and 90, 97
    fixed- and variable-length 90

fixed-length 97
unindexed 22
variable-length 97
configuration (server)
number of rows per page 77
contention
disk I/O 4
I/O device 4
logical devices and 3
max_rows_per_page and 76
partitions to avoid 11
transaction log writes 47
underlying problems 3
controller, device 4
covered queries
index covering 21
covering nonclustered indexes
rebuilding 109
create clustered index command
sorted_data and **fillfactor** interaction 61
sorted_data and **reservepagegap** interaction 73–75
create index command
fillfactor and 56–61
locks acquired by 106
reservepagegap option 70
segments and 107
sorted_data option 107
create table command
exp_row_size option 63
reservepagegap option 69
space management properties 63

D

data
max_rows_per_page and storage 76
storage 4, 21–47
data caches
aging in 49
binding objects to 49
data modification and 51
deletes on heaps and 53
fetch-and-discard strategy 50
inserts to heaps and 52
joins and 50
tempdb bound to own 135
updates to heaps and 53
wash marker 49
data modification
data caches and 51
heap tables and 41
log space and 113
transaction log and 47
data pages 23–47
computing number of 91, 98
fillfactor effect on 58
limiting number of rows on 76
linking 39
partially full 46
text and image 25
data, inserting into an allpages locked table 41
database devices 1
parallel queries and 5
sybsecurity 6
tempdb 6
database objects
binding to caches 49
placement 1–19
placement on segments 2
storage 21–47
databases
creation speed 110
devices and 4
placement 2
dbcc tune
cleanup 117
des _bind 117
default exp_row_size percent configuration parameter 64
default fill factor percentage configuration parameter 59
default settings
max_rows_per_page 77
delete operations
heap tables 43
object size and 81
denormalization
temporary tables and 136
devices
adding for partitioned tables 15
object placement on 2

partitioned tables and 15
throughput, measuring 12
disk devices
performance and 1–19
disk mirroring
device placement 7
performance and 3
DOL columns
wide, variable length 34–36

E

exceed logical page size 33
exp_row_size option 62–67
create table 63
default value 63
server-wide default 64
setting before **alter table...lock** 122
sp_chgattribute 64
storage required by 102
expected row size. See **exp_row_size** option
extents
allocation and **reservepagegap** 68
space allocation and 25

F

fetch-and-discard cache strategy 50
fillfactor
advantages of 56
disadvantages of 57
index page size and 58
locking and 76
max_rows_per_page compared to 76
page splits and 56
fillfactor option
See also **fillfactor** values
create index 56
sorted_data option and 61
fillfactor values
See fillfactor option
fillfactor values
alter table...lock 59
applied to data pages 60
applied to index pages 60
clustered index creation and 59
nonclustered index rebuilds 59
reorg rebuild 59
table-level 59
first page
allocation page 27
text pointer 25
fixed-length columns
calculating space for 86
data row size of 90, 97
index row size and 91
for load option
performance and 110
formulas
table or index sizes 86–104
forwarded rows
query on systabstats 66
reserve page gap and 68
fragmentation, reserve page gap and 68

G

global allocation map (GAM) pages 27

H

hardware
terminology 1
hash-based scans
joins and 5
header information
data pages 24
heap tables 38–47
bcp (bulk copy utility) and 116
delete operations 43
deletes and pages in cache 53
guidelines for using 39
I/O and 45
I/O inefficiency and 46
insert operations on 41
inserts and pages in cache 52
locking 41
maintaining 46

performance limits 41
select operations on 40, 51
updates and pages in cache 53
updates on 44

I

I/O
access problems and 3
balancing load with segments 10
bcp (bulk copy utility) and 116
create database and 110
default caches and 49
devices and 2
efficiency on heap tables 46
expected row size and 67
heap tables and 45
increasing size of 45
performance and 4
recovery interval and 113
select operations on heap tables and 51
server-wide and database 5
sp_spaceused and 83
spreading between caches 135
transaction log and 47
image datatype
page size for storage 25
storage on separate device 10, 25
index covering
definition 21
index pages
fillfactor effect on 57, 58
limiting number of rows on 76
indexes
access through 21
bulk copy and 113
choosing 22
computing number of pages 92
creating 106
max_rows_per_page and 77
rebuilding 109
recovery and creation 107
size of 80
sort order changes 109
sp_spaceused size report 83
temporary tables and 138
usefulness of 39
initializing
text or image pages 103
insert operations
heap tables and 41
logging and 137
partitions and 11
performance of 3
rebuilding indexes after many 109

J

joins
data cache and 50
hash-based scan and 5
temporary tables for 136

L

large object (LOB) 10
leaf levels of indexes
fillfactor and number of rows 58
queries on 22
row size calculation 94, 98
leaf pages
calculating number in index 94, 99
limiting number of rows on 76
load balancing for partitioned tables
maintaining 18
local backups 112
locking
create index and 106
heap tables and inserts 41
logging
bulk copy and 113
minimizing in tempdb 136
logical device name 1
logical page sizes 23
LRU replacement strategy 49, 50

M

maintenance tasks 105–117
  performance and 3
map, object allocation. See object allocation map (OAM) pages
max_rows_per_page option
  fillfactor compared to 76
  locking and 76
  select into effects 77
modes of disk mirroring 7
MRU replacement strategy 49

N

nesting
  temporary tables and 139
networks
  reducing traffic on 117
nonclustered indexes
  estimating size of 94–96
  size of 83, 94, 98
nonleaf rows 95
normalization
  temporary tables and 136
null columns
  storage of rows 24
  storage size 88
null values
  text and image columns 103
number (quantity of)
  OAM pages 96, 100
  rows (rowtotal), estimated 82
  rows on a page 76
number of columns and sizes 31

O

object allocation map (OAM) pages 27
  LRU strategy in data cache 49
  overhead calculation and 93, 98
object size
  viewing with **optdiag** 81
offset table
  size of 24
online backups 112
optdiag utility command
  object sizes and 81
optimizer
  temporary tables and 137
order
  presorted data and index creation 107
  recovery of databases 113
  result sets and performance 39
output
  sp_spaceused 82
overhead 30
  calculation (space allocation) 96, 100
  clustered indexes and 39
  object size calculations 86
  row and page 86
  space allocation calculation 93, 98
  variable-length and null columns 88

P

page chains
  placement 2
  text or *image* data 103
page splits
  fillfactor effect on 56
  max_rows_per_page setting and 76
  object size and 81
  reducing 56
page utilization percent configuration parameter
  object size estimation and 87
pages
  global allocation map (GAM) 27
pages, data 23–47
  bulk copy and allocations 113
  calculating number of 91, 98
  fillfactor effect on 58
  linking 39
  size 23
pages, index
  calculating number of 92
  calculating number of nonleaf 99
  fillfactor effect on 57, 58
pages, OAM (object allocation map) 27
  number of 93, 96, 98, 100

parallel query processing
  object placement and 2
  performance of 3
partitioned tables 11
  bcp (bulk copy utility) and 116
  devices and 15
  maintaining 18
  read-mostly 13
  read-only 13
  space planning for 12
  updates and 14
performance
  backups and 112
  bcp (bulk copy utility) and 115
  clustered indexes and 39
  tempdb and 139
physical device name 1
point query 22
pointers
  last page, for heap tables 41
  page chain 39
  text and image page 25
pools, data cache
  configuring for operations on heap tables 45
precision, datatype
  size and 88
prefetch
  sequential 45

Q

queries
  point 22
  unindexed columns in 22

R

RAID devices
  partitioned tables and 12
reads
  disk mirroring and 7
  image values 25
  text values 25
recovery
  index creation and 107
  log placement and speed 6
recovery interval in minutes configuration parameter
  I/O and 113
recreating
  indexes 107
remote backups 112
replacement strategy. *See* LRU replacement strategy; MRU replacement strategy
reports
  sp_estspace 85
reserved pages, **sp_spaceused** report on 84
reservepagegap option 68–73
  cluster ratios 68, 73
  create index 70
  create table 69
  extent allocation and 68
  forwarded rows and 68
  sp_chgattribute 70
  space usage and 68
  storage required by 102
response time
  table scans and 22
rounding
  object size calculation and 87
row offsets 34
rows per data page 37
rows, index
  size of leaf 94, 98
  size of nonleaf 95

S

scans, table
  performance issues 22
segments 1
  changing table locking schemes 120
  clustered indexes on 10
  database object placement on 5, 10
  nonclustered indexes on 10
  tempdb 132
select * command
  logging of 137
select into command
  heap tables and 41

select operations
heaps 40
sequential prefetch 45
size
data pages 23
datatypes with precisions 88
formulas for tables or indexes 86–104
I/O 45
indexes 80
object (**sp_spaceused**) 82
predicting tables and indexes 89–104
sp_spaceused estimation 84
tables 80
tempdb database 132
sort operations (**order by**)
improving performance of 106
performance problems 126
sort order
rebuilding indexes after changing 109
sorted data, reindexing 107
sorted_data option
fillfactor and 61
reservepagegap and 73
sorted_data option, **create index**
sort suppression and 107
sp_chgattribute system procedure
fillfactor 57
sp_chgattribute system procedure
exp_row_size 64
reservepagegap 70
sp_estspace system procedure
advantages of 86
disadvantages of 86
planning future growth with 84
sp_help system procedure
displaying expected row size 65
sp_spaceused system procedure 82
row total estimate reported 82
space 30, 31
estimating table and index size 89–104
extents 25
for *text* or *image* storage 25
reclaiming 46
unused 26
space allocation
contiguous 29
deletes and 43
extents 25
object allocation map (OAM) pages 93, 98
overhead calculation 93, 96, 98, 100
predicting tables and indexes 89–104
sp_spaceused 84
tempdb 134
unused space within 26
space management properties 55–78
object size and 101
reserve page gap 68–73
space usage 122
speed (server)
select into 137
sort operations 106
storage management
delete operations and 43
I/O contention avoidance 4
page proximity 29
row storage 24
stored procedures
performance and 3
temporary tables and 139
striping tempdb 132
sybsecurity database
placement 6
sysgams table 27
sysindexes table
data access and 29
text objects listed in 25
system tables
data access and 29
performance and 3

T

table scans
performance issues 22
tables
estimating size of 86
heap 38–47
size of 80
size with a clustered index 89, 96
tempdb database
data caches 135

logging in 136
performance and 139
placement 6, 131
segments 132
space allocation 134
striping 132
temporary tables
denormalization and 136
indexing 138
nesting procedures and 139
normalization and 136
optimizing 137
performance considerations 3
text datatype
chain of text pages 103
page size for storage 25
storage on separate device 10, 25
sysindexes table and 25
thresholds
bulk copy and 115
database dumps and 113
throughput
measuring for devices 12
transaction logs
placing on separate segment 6
on same device 7
storage as heap 47
transactions
logging and 137

U

units, allocation. *See* allocation units
unused space
allocations and 26
update command
image data and 103
text data and 103
update operations
heap tables and 44

V

variable-length 33
variable-length rows, wide 34

W

wash marker 49
where clause
table scans and 39
wide, variable length DOL columns 34–36
BCP 35
downgrade 35
dump and loading 35
proxy tables 35
write operations
disk mirroring and 7
image values 25
serial mode of disk mirroring 8
text values 25