



Performance and Tuning: Basics

**Adaptive Server® Enterprise
12.5.1**

DOCUMENT ID: DC20020-01-1251-01

LAST REVISED: August 2003

Copyright © 1989-2003 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, the Sybase logo, AccelaTrade, ADA Workbench, Adaptable Windowing Environment, Adaptive Component Architecture, Adaptive Server, Adaptive Server Anywhere, Adaptive Server Enterprise, Adaptive Server Enterprise Monitor, Adaptive Server Enterprise Replication, Adaptive Server Everywhere, Adaptive Server IQ, Adaptive Warehouse, Anywhere Studio, Application Manager, AppModeler, APT Workbench, APT-Build, APT-Edit, APT-Execute, APT-FORMS, APT-Translator, APT-Library, AvantGo, AvantGo Application Alerts, AvantGo Mobile Delivery, AvantGo Mobile Document Viewer, AvantGo Mobile Inspection, AvantGo Mobile Marketing Channel, AvantGo Mobile Pharma, AvantGo Mobile Sales, AvantGo Pylon, AvantGo Pylon Application Server, AvantGo Pylon Conduit, AvantGo Pylon PIM Server, AvantGo Pylon Pro, Backup Server, BizTracker, ClearConnect, Client-Library, Client Services, Convoy/DM, Copernicus, Data Pipeline, Data Workbench, DataArchitect, Database Analyzer, DataExpress, DataServer, DataWindow, DB-Library, dbQueue, Developers Workbench, Direct Connect Anywhere, DirectConnect, Distribution Director, e-ADK, E-Anywhere, e-Biz Integrator, E-Whatever, EC Gateway, ECMAP, ECRTP, eFulfillment Accelerator, Embedded SQL, EMS, Enterprise Application Studio, Enterprise Client/Server, Enterprise Connect, Enterprise Data Studio, Enterprise Manager, Enterprise SQL Server Manager, Enterprise Work Architecture, Enterprise Work Designer, Enterprise Work Modeler, eProcurement Accelerator, EWA, Financial Fusion, Financial Fusion Server, Gateway Manager, GlobalFIX, ImpactNow, Industry Warehouse Studio, InfoMaker, Information Anywhere, Information Everywhere, InformationConnect, InternetBuilder, iScript, Jaguar CTS, jConnect for JDBC, Mail Anywhere Studio, MainframeConnect, Maintenance Express, Manage Anywhere Studio, M-Business Channel, M-Business Network, M-Business Server, MDI Access Server, MDI Database Gateway, media.splash, MetaWorks, My AvantGo, My AvantGo Media Channel, My AvantGo Mobile Marketing, MySupport, Net-Gateway, Net-Library, New Era of Networks, ObjectConnect, ObjectCycle, OmniConnect, OmniSQL Access Module, OmniSQL Toolkit, Open Biz, Open Client, Open ClientConnect, Open Client/Server, Open Client/Server Interfaces, Open Gateway, Open Server, Open ServerConnect, Open Solutions, Optima++, PB-Gen, PC APT Execute, PC Net Library, PocketBuilder, Pocket PowerBuilder, Power++, power.stop, PowerAMC, PowerBuilder, PowerBuilder Foundation Class Library, PowerDesigner, PowerDimensions, PowerDynamo, PowerJ, PowerScript, PowerSite, PowerSocket, Powersoft, PowerStage, PowerStudio, PowerTips, Powersoft Portfolio, Powersoft Professional, PowerWare Desktop, PowerWare Enterprise, ProcessAnalyst, Rapport, Report Workbench, Report-Execute, Replication Agent, Replication Driver, Replication Server, Replication Server Manager, Replication Toolkit, Resource Manager, RW-DisplayLib, S-Designer, SDF, Secure SQL Server, Secure SQL Toolset, Security Guardian, SKILS, smart.partners, smart.parts, smart.script, SQL Advantage, SQL Anywhere, SQL Anywhere Studio, SQL Code Checker, SQL Debug, SQL Edit, SQL Edit/TPU, SQL Everywhere, SQL Modeler, SQL Remote, SQL Server, SQL Server Manager, SQL SMART, SQL Toolset, SQL Server/CFT, SQL Server/DBM, SQL Server SNMP SubAgent, SQL Station, SQLJ, STEP, SupportNow, S.W.I.F.T. Message Format Libraries, Sybase Central, Sybase Client/Server Interfaces, Sybase Financial Server, Sybase Gateways, Sybase MPP, Sybase SQL Desktop, Sybase SQL Lifecycle, Sybase SQL Workgroup, Sybase User Workbench, SybaseWare, Syber Financial, SyberAssist, SyBooks, System 10, System 11, System XI (logo), SystemTools, Tabular Data Stream, TradeForce, Transact-SQL, Translation Toolkit, UltraLite.NET, UNIBOM, Unilib, Uninull, Unisep, Unistring, URK Runtime Kit for UniCode, Viewer, Visual Components, VisualSpeller, VisualWriter, VQL, WarehouseArchitect, Warehouse Control Center, Warehouse Studio, Warehouse WORKS, Watcom, Watcom SQL, Watcom SQL Server, Web Deployment Kit, Web.PB, Web.SQL, WebSights, WebViewer, WorkGroup SQL Server, XA-Library, XA-Server and XP Server are trademarks of Sybase, Inc. 03/03

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

All other company and product names used herein may be trademarks or registered trademarks of their respective companies.

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

About This Book	xv
CHAPTER 1 Introduction to Performance and Tuning	1
CHAPTER 2 Introduction to the Basics	3
Good performance	3
Response time	3
Throughput	4
Designing for performance	4
Tuning performance	4
Tuning levels	5
Configuration parameters	10
Dynamic	11
Identifying system limits	12
Varying logical page sizes	12
Number of columns and column size	12
Maximum length of expressions, variables, and stored procedure arguments	13
Number of logins	13
Performance implications for limits	14
Setting tuning goals	14
Analyzing performance	14
Normal Forms	15
Locking	16
Special Considerations	16
CHAPTER 3 Networks and Performance	19
Introduction	19
Potential performance problems	19
Basic questions on network performance	20
Techniques summary	20
Using sp_sysmon while changing network configuration	21
How Adaptive Server uses the network	21

- Managing Network Listeners..... 21
 - Network Listeners on UNIX 22
 - Managing listeners with sp_listener 23
 - Using the remaining parameter 24
 - Determining the status of listeners 24
 - Starting new listeners 25
 - Stopping listeners 26
 - Suspending listeners 26
 - Resume suspended listeners 27
- Changing network packet sizes 27
- Large versus default packet sizes for user connections 28
 - Number of packets is important..... 28
 - Evaluation tools with Adaptive Server 29
 - Evaluation tools outside of Adaptive Server..... 30
 - Server-based techniques for reducing network traffic 30
- Impact of other server activities 31
 - Single user versus multiple users..... 32
- Improving network performance..... 32
 - Isolate heavy network users 32
 - Set tcp no delay on TCP networks 33
 - Configure multiple network listeners 34

CHAPTER 4 Using Engines and CPUs..... 35

- Background concepts..... 35
 - How Adaptive Server processes client requests 36
 - Client task implementation 37
- Single-CPU process model 38
 - Scheduling engines to the CPU 38
 - Scheduling tasks to the engine 40
 - Execution task scheduling 41
- Adaptive Server SMP process model 43
 - Scheduling engines to CPUs..... 44
 - Scheduling Adaptive Server tasks to engines 44
 - Multiple network engines 45
 - Task priorities and run queues 45
 - Processing scenario 46
- Asynchronous log service 47
 - Understanding the user log cache (ULC) architecture 48
 - When to use ALS 48
 - Using the ALS 49
- Housekeeper task improves CPU utilization 50
 - Side effects of the housekeeper task 51
 - Configuring the housekeeper task..... 51
- Measuring CPU usage 53

Single-CPU machines	53
Determining when to configure additional engines.....	54
Taking engines offline	55
Enabling engine-to-CPU affinity	55
Multiprocessor application design guidelines.....	57

CHAPTER 5 Distributing Engine Resources..... 59

Algorithm for successfully distributing engine resources	59
Algorithm guidelines	62
Environment analysis and planning.....	63
Performing benchmark tests	65
Setting goals.....	66
Results analysis and tuning.....	66
Monitoring the environment over time	66
Manage preferred access to resources.....	67
Types of execution classes	67
Predefined execution classes.....	68
User-Defined execution classes.....	68
Execution class attributes	69
Base priority	69
Time slice	70
Task-to-engine affinity	70
Setting execution class attributes.....	71
Assigning execution classes	72
Engine groups and establishing task-to-engine affinity	72
How execution class bindings affect scheduling	74
Setting attributes for a session only	76
Getting information	76
Rules for determining precedence and scope.....	77
Multiple execution objects and ECs	77
Resolving a precedence conflict.....	80
Examples: determining precedence	80
Example scenario using precedence rules	82
Planning	83
Configuration	84
Execution characteristics.....	85
Considerations for Engine Resource Distribution	85
Client applications: OLTP and DSS	86
Adaptive Server logins: high-priority users.....	87
Stored procedures: “hot spots”.....	87

CHAPTER 6 Controlling Physical Data Placement..... 89

Object placement can improve performance	89
--	----

- Symptoms of poor object placement 90
- Underlying problems 91
- Using sp_sysmon while changing data placement..... 91
- Terminology and concepts 92
- Guidelines for improving I/O performance 92
 - Spreading data across disks to avoid I/O contention 93
 - Isolating server-wide I/O from database I/O..... 94
 - Keeping transaction logs on a separate disk..... 94
 - Mirroring a device on a separate disk 95
- Creating objects on segments..... 96
 - Using segments..... 97
 - Separating tables and indexes 98
 - Splitting large tables across devices 98
 - Moving text storage to a separate device..... 98
- Partitioning tables for performance 99
 - User transparency 99
 - Partitioned tables and parallel query processing..... 100
 - Improving insert performance with partitions..... 101
 - Restrictions on partitioned tables 102
 - Partition-related configuration parameters 102
 - How Adaptive Server distributes partitions on devices 102
- Space planning for partitioned tables 103
 - Read-only tables 104
 - Read-mostly tables..... 105
 - Tables with random data modification..... 105
- Commands for partitioning tables 106
 - alter table...partition syntax 106
 - alter table...unpartition Syntax..... 107
 - Changing the number of partitions 107
 - Distributing data evenly across partitions..... 108
 - Using parallel bcp to copy data into partitions..... 110
 - Getting information about partitions 111
 - Using bcp to correct partition balance 112
 - Checking data distribution on devices with sp_helpsegment 114
 - Updating partition statistics 115
- Steps for partitioning tables..... 117
 - Backing up the database after partitioning tables 117
 - Table does not exist 117
 - Table exists elsewhere in the database 119
 - Table exists on the segment 119
- Special procedures for difficult situations 124
 - Clustered indexes on large tables 124
 - Alternative for clustered indexes 125
- Problems when devices for partitioned tables are full 128

Adding disks when devices are full 128
 Adding disks when devices are nearly full..... 130
 Maintenance issues and partitioned tables 131
 Regular maintenance checks for partitioned tables 131

CHAPTER 7

Database Design 133
 Basic design 133
 Physical database design for Adaptive Server 134
 Logical Page Sizes 134
 Number of columns and column size 135
 Normalization 135
 Levels of normalization 136
 Benefits of normalization 136
 First Normal Form 137
 Second Normal Form 138
 Third Normal Form 139
 Denormalizing for performance 141
 Risks 142
 Denormalization input 143
 Techniques 144
 Splitting tables 146
 Managing denormalized data 148
 Using triggers 149
 Using application logic 149
 Batch reconciliation 150

CHAPTER 8

Data Storage 151
 Performance gains through query optimization 151
 Query processing and page reads 152
 Adaptive Server pages 153
 Page headers and page sizes 154
 Varying logical page sizes 154
 Data and index pages 155
 Large Object (LOB) Pages 156
 Extents 156
 Pages that manage space allocation 157
 Global allocation map pages 157
 Allocation pages 158
 Object allocation map pages 158
 How OAM pages and allocation pages manage object storage 158
 Page allocation keeps an object's pages together 159
 sysindexes table and data access 160
 Space overheads 160

- Number of columns and size..... 161
- Number of rows per data page..... 165
- Maximum numbers..... 166
- Heaps of data: tables without clustered indexes..... 167
 - Lock schemes and differences between heaps 167
 - Select operations on heaps..... 168
 - Inserting data into an allpages-locked heap table..... 168
 - Inserting data into a data-only-locked heap table..... 169
 - Deleting data from a heap table 170
 - Updating data on a heap table 171
- How Adaptive Server performs I/O for heap operations 172
 - Sequential prefetch, or large I/O 173
- Caches and object bindings 174
 - Heaps, I/O, and cache strategies..... 174
 - Select operations and caching 176
 - Data modification and caching 177
- Asynchronous prefetch and I/O on heap tables 179
- Heaps: pros and cons 180
- Maintaining heaps 180
 - Methods..... 180
- Transaction log: a special heap table..... 181

CHAPTER 9 Setting Space Management Properties 183

- Reducing index maintenance..... 183
 - Advantages of using fillfactor 184
 - Disadvantages of using fillfactor..... 184
 - Setting fillfactor values 185
 - fillfactor examples..... 185
 - Use of the sorted_data and fillfactor options 188
- Reducing row forwarding 189
 - Default, minimum, and maximum values for exp_row_size .. 189
 - Specifying an expected row size with create table..... 190
 - Adding or changing an expected row size..... 191
 - Setting a default expected row size server-wide 191
 - Displaying the expected row size for a table..... 192
 - Choosing an expected row size for a table 192
 - Conversion of max_rows_per_page to exp_row_size..... 193
 - Monitoring and managing tables that use expected row size 194
- Leaving space for forwarded rows and inserts..... 194
 - Extent allocation operations and reservepagegap 195
 - Specifying a reserve page gap with create table..... 196
 - Specifying a reserve page gap with create index..... 197
 - Changing reservepagegap 197
 - reservepagegap examples 198

Choosing a value for reservepagegap 199
 Monitoring reservepagegap settings 199
 reservepagegap and sorted_data options to create index 200
 Using max_rows_per_page on allpages-locked tables 202
 Reducing lock contention 203
 Indexes and max_rows_per_page 203
 select into and max_rows_per_page..... 204
 Applying max_rows_per_page to existing data..... 204

CHAPTER 10

Memory Use and Performance 205
 How memory affects performance 205
 How much memory to configure 206
 Dynamic reconfiguration 209
 Dynamic memory allocation 209
 How memory is allocated 210
 Caches in Adaptive Server..... 211
 CAche sizes and buffer pools..... 211
 Procedure cache 212
 Getting information about the procedure cache size..... 212
 Procedure cache sizing 213
 Estimating stored procedure size 214
 Data cache 215
 Default cache at installation time..... 215
 Page aging in data cache..... 215
 Effect of data cache on retrievals 216
 Effect of data modifications on the cache..... 217
 Data cache performance 218
 Testing data cache performance..... 218
 Configuring the data cache to improve performance 220
 Commands to configure named data caches 222
 Tuning named caches 222
 Cache configuration goals 223
 Gather data, plan, and then implement 224
 Evaluating cache needs 225
 Large I/O and performance 225
 Reducing spinlock contention with cache partitions 228
 Cache replacement strategies and policies..... 228
 Named data cache recommendations 230
 Sizing caches for special objects, tempdb, and transaction logs .
 232
 Basing data pool sizes on query plans and I/O 236
 Configuring buffer wash size 238
 Overhead of pool configuration and binding objects 239
 Maintaining data cache performance for large I/O 240

Diagnosing excessive I/O Counts	241
Using sp_sysmon to check large I/O performance.....	241
Speed of recovery	242
Tuning the recovery interval	242
Effects of the housekeeper wash task on recovery time	243
Auditing and performance	243
Sizing the audit queue	244
Auditing performance guidelines	245
Text and images pages	245

CHAPTER 11

Determining Sizes of Tables and Indexes	247
Why object sizes are important to query tuning	247
Tools for determining the sizes of tables and indexes	248
Effects of data modifications on object sizes	249
Using optdiag to display object sizes	249
Advantages of optdiag.....	250
Disadvantages of optdiag.....	250
Using sp_spaceused to display object size.....	250
Advantages of sp_spaceused	251
Disadvantages of sp_spaceused	252
Using sp_estspace to estimate object size	252
Advantages of sp_estspace	253
Disadvantages of sp_estspace	254
Using formulas to estimate object size.....	254
Factors that can affect storage size	254
Storage sizes for datatypes.....	255
Tables and indexes used in the formulas.....	257
Calculating table and clustered index sizes for allpages-locked tables	257
Calculating the sizes of data-only-locked tables	263
Other factors affecting object size	268
Very small rows	269
LOB pages	270
Advantages of using formulas to estimate object size	271
Disadvantages of using formulas to estimate object size.....	271

CHAPTER 12

How Indexes Work.....	273
Types of indexes	274
Index pages.....	274
Index Size.....	276
Clustered indexes on allpages-locked tables.....	276
Clustered indexes and select operations	277
Clustered indexes and insert operations	278

Page splitting on full data pages	279
Page splitting on index pages	281
Performance impacts of page splitting	281
Overflow pages	282
Clustered indexes and delete operations	283
Nonclustered indexes.....	285
Leaf pages revisited	285
Nonclustered index structure.....	286
Nonclustered indexes and select operations.....	287
Nonclustered index performance	288
Nonclustered indexes and insert operations	289
Nonclustered indexes and delete operations	290
Clustered indexes on data-only-locked tables.....	291
Index covering.....	291
Covering matching index scans	292
Covering nonmatching index scans	293
Indexes and caching	295
Using separate caches for data and index pages	295
Index trips through the cache	295

CHAPTER 13	Indexing for Performance	297
	How indexes affect performance.....	297
	Detecting indexing problems.....	298
	Symptoms of poor indexing.....	298
	Fixing corrupted indexes	301
	Repairing the system table index	301
	Index limits and requirements	304
	Choosing indexes.....	305
	Index keys and logical keys.....	306
	Guidelines for clustered indexes	306
	Choosing clustered indexes	307
	Candidates for nonclustered indexes	307
	Index Selection.....	308
	Other indexing guidelines.....	310
	Choosing nonclustered indexes	311
	Choosing composite indexes	312
	Key order and performance in composite indexes	312
	Advantages and disadvantages of composite indexes	314
	Techniques for choosing indexes.....	315
	Choosing an index for a range query	315
	Adding a point query with different indexing requirements....	316
	Index and statistics maintenance	317
	Dropping indexes that hurt performance	318
	Choosing space management properties for indexes	318

- Additional indexing tips 319
 - Creating artificial columns 319
 - Keeping index entries short and avoiding overhead 319
 - Dropping and rebuilding indexes 320
 - Configure enough sort buffers 320
 - Create the clustered index first 320
 - Configure large buffer pools 320
- Asynchronous log service 320
 - Understanding the user log cache (ULC) architecture 322
 - When to use ALS 322
 - Using the ALS 323

CHAPTER 14 Cursors and Performance..... 325

- Definition 325
 - Set-oriented versus row-oriented programming 326
 - Example 327
- Resources required at each stage 328
 - Memory use and execute cursors 330
- Cursor modes 331
- Index use and requirements for cursors 331
 - Allpages-locked tables 331
 - Data-only-locked tables 332
- Comparing performance with and without cursors 333
 - Sample stored procedure without a cursor 333
 - Sample stored procedure with a cursor 334
 - Cursor versus noncursor performance comparison 335
- Locking with read-only cursors 336
- Isolation levels and cursors 338
- Partitioned heap tables and cursors 338
- Optimizing tips for cursors 339
 - Optimizing for cursor selects using a cursor 339
 - Using union instead of or clauses or in lists 340
 - Declaring the cursor's intent 340
 - Specifying column names in the for update clause 340
 - Using set cursor rows 341
 - Keeping cursors open across commits and rollbacks 342
 - Opening multiple cursors on a single connection 342

CHAPTER 15 Maintenance Activities and Performance..... 343

- Running reorg on tables and indexes 343
- Creating and maintaining indexes 344
 - Configuring Adaptive Server to speed sorting 344
 - Dumping the database after creating an index 345

Creating an index on sorted data.....	345
Maintaining index and column statistics	346
Rebuilding indexes	347
Creating or altering a database	348
Backup and recovery.....	350
Local backups.....	350
Remote backups.....	350
Online backups.....	351
Using thresholds to prevent running out of log space.....	351
Minimizing recovery time	351
Recovery order	351
Bulk copy	352
Parallel bulk copy.....	352
Batches and bulk copy.....	353
Slow bulk copy.....	353
Improving bulk copy performance.....	353
Replacing the data in a large table	354
Adding large amounts of data to a table	354
Using partitions and multiple bulk copy processes	354
Impacts on other users	355
Database consistency checker.....	355
Using dbcc tune (cleanup).....	355
Using dbcc tune on spinlocks	356
When not to use this command	356
Determining the space available for maintenance activities.....	356
Overview of space requirements	357
Tools for checking space usage and space available	358
Estimating the effects of space management properties.....	360
If there is not enough space	361

CHAPTER 16

Tuning Asynchronous Prefetch	363
How asynchronous prefetch improves performance	363
Improving query performance by prefetching pages	364
Prefetching control mechanisms in a multiuser environment	365
Look-ahead set during recovery	366
Look-ahead set during sequential scans	366
Look-ahead set during nonclustered index access.....	367
Look-ahead set during dbcc checks	367
Look-ahead set minimum and maximum sizes.....	368
When prefetch is automatically disabled	369
Flooding pools	370
I/O system overloads	370
Unnecessary reads.....	371
Tuning Goals for asynchronous prefetch.....	373

Commands for configuration.....	374
Other Adaptive Server performance features.....	374
Large I/O.....	374
Fetch-and-discard (MRU) scans.....	376
Parallel scans and large I/Os.....	376
Special settings for asynchronous prefetch limits.....	377
Setting limits for recovery	377
Setting limits for dbcc.....	378
Maintenance activities for high prefetch performance	378
Eliminating kinks in heap tables.....	379
Eliminating kinks in clustered index tables	379
Eliminating kinks in nonclustered indexes	379
Performance monitoring and asynchronous prefetch.....	379
CHAPTER 17	
tempdb Performance Issues	381
How management of tempdb affects performance.....	381
Main solution areas for tempdb performance	382
Types and uses of temporary tables	382
Truly temporary tables	383
Regular user tables.....	383
Worktables.....	384
Initial allocation of tempdb	384
Sizing the tempdb.....	385
Placing tempdb.....	386
Dropping the master device from tempdb segments.....	386
Using multiple disks for parallel query performance	387
Binding tempdb to its own cache	387
Commands for cache binding	388
Temporary tables and locking	388
Minimizing logging in tempdb	389
With select into	389
By using shorter rows	389
Optimizing temporary tables.....	390
Creating indexes on temporary tables.....	391
Creating nested procedures with temporary tables	391
Breaking tempdb uses into multiple procedures.....	392
Index.....	393

About This Book

Audience

This manual is intended for database administrators, database designers, developers and system administrators.

Note You may want to use your own database for testing changes and queries. Take a snapshot of the database in question and set it up on a test machine.

How to use this book

Chapter 1, “Introduction to Performance and Tuning” .

Chapter 2, “Introduction to the Basics” describes the major components to be analyzed when addressing performance.

Chapter 3, “Networks and Performance” provides a brief description of relational databases and good database design.

Chapter 4, “Using Engines and CPUs” describes how client processes are scheduled on engines in Adaptive Server.

Chapter 5, “Distributing Engine Resources” describes how to assign execution precedence to specific applications.

Chapter 6, “Controlling Physical Data Placement” describes the uses of segments and partitions for controlling the physical placement of data on storage devices.

Chapter 7, “Database Design” provides a brief description of relational databases and good database design.

Chapter 8, “Data Storage” describes Adaptive Server page types, how data is stored on pages, and how queries on heap tables are executed

Chapter 9, “Setting Space Management Properties” describes how space management properties can be set for tables to improve performance and reduce the frequency of maintenance operations on tables and indexes.

Chapter 10, “Memory Use and Performance” describes how Adaptive Server uses memory for the procedure and data caches.

Chapter 11, “Determining Sizes of Tables and Indexes” describes different methods for determining the current size of database objects and for estimating their future size.

Chapter 15, “Maintenance Activities and Performance” describes the impact of maintenance activities on performance, and how some activities, such as re-creating indexes, can improve performance.

Related documents

- The remaining manuals for the Performance and Tuning Series are:
 - *Performance and Tuning: Locking*
 - *Performance and Tuning: Monitoring and Analyzing*
 - *Performance and Tuning: Optimizer and Abstract Plans*
- The release bulletin for your platform – contains last-minute information that was too late to be included in the books.

A more recent version of the release bulletin may be available on the World Wide Web. To check for critical product or document information that was added after the release of the product CD, use the Sybase Technical Library.

- *The Installation Guide* for your platform – describes installation, upgrade, and configuration procedures for all Adaptive Server and related Sybase products.
- *Configuring Adaptive Server Enterprise* for your platform – provides instructions for performing specific configuration tasks for Adaptive Server.
- *What’s New in Adaptive Server Enterprise?* – describes the new features in Adaptive Server version 12.5, the system changes added to support those features, and the changes that may affect your existing applications.
- *Transact-SQL User’s Guide* – documents Transact-SQL, Sybase’s enhanced version of the relational database language. This manual serves as a textbook for beginning users of the database management system. This manual also contains descriptions of the pubs2 and pubs3 sample databases.
- *System Administration Guide* – provides in-depth information about administering servers and databases. This manual includes instructions and guidelines for managing physical resources, security, user and system databases, and specifying character conversion, international language, and sort order settings.

- *Reference Manual* – contains detailed information about all Transact-SQL commands, functions, procedures, and data types. This manual also contains a list of the Transact-SQL reserved words and definitions of system tables.
- The *Utility Guide* – documents the Adaptive Server utility programs, such as isql and bcp, which are executed at the operating system level.
- The *Quick Reference Guide* – provides a comprehensive listing of the names and syntax for commands, functions, system procedures, extended system procedures, data types, and utilities in a pocket-sized book. Available only in print version.
- The *System Tables Diagram* – illustrates system tables and their entity relationships in a poster format. Available only in print version.
- *Error Messages and Troubleshooting Guide* – explains how to resolve frequently occurring error messages and describes solutions to system problems frequently encountered by users.
- *Component Integration Services User's Guide* – explains how to use the Adaptive Server Component Integration Services feature to connect remote Sybase and non-Sybase databases.
- *Java in Adaptive Server Enterprise* – describes how to install and use Java classes as data types, functions, and stored procedures in the Adaptive Server database.
- *XML Services in Adaptive Server Enterprise* – describes the Sybase native XML processor and the Sybase Java-based XML support, introduces XML in the database, and documents the query and mapping functions that comprise XML Services.
- *Using Sybase Failover in a High Availability System* – provides instructions for using Sybase's Failover to configure an Adaptive Server as a companion server in a high availability system.
- *Job Scheduler User's Guide* – provides instructions on how to create and schedule jobs on a local or remote Adaptive Server using the command line or a graphical user interface (GUI).
- *Using Adaptive Server Distributed Transaction Management Features* – explains how to configure, use, and troubleshoot Adaptive Server DTM features in distributed transaction processing environments.
- *EJB Server User's Guide* – explains how to use EJB Server to deploy and execute Enterprise JavaBeans in Adaptive Server.

-
- *XA Interface Integration Guide for CICS, Encina, and TUXEDO* – provides instructions for using Sybase’s DTM XA interface with X/Open XA transaction managers.
 - *Glossary* – defines technical terms used in the Adaptive Server documentation.
 - *Sybase jConnect for JDBC Programmer’s Reference* – describes the jConnect for JDBC product and explains how to use it to access data stored in relational database management systems.
 - *Full-Text Search Specialty Data Store User’s Guide* – describes how to use the Full-Text Search feature with Verity to search Adaptive Server Enterprise data.
 - *Historical Server User’s Guide* –describes how to use Historical Server to obtain performance information for SQL Server and Adaptive Server.
 - *Monitor Server User’s Guide* – describes how to use Monitor Server to obtain performance statistics from SQL Server and Adaptive Server.
 - *Monitor Client Library Programmer’s Guide* – describes how to write Monitor Client Library applications that access Adaptive Server performance data.

Other sources of information

Use the Sybase Technical Library CD and the Technical Library Product Manuals Web site to learn more about your product:

- The Getting Started CD contains release bulletins and installation guides in PDF format, and may also contain other documents or updated information not included on the Technical Library CD. It is included with your software. To read or print documents on the Getting Started CD you need Adobe Acrobat Reader (downloadable at no charge from the Adobe Web site, using a link provided on the CD).
- The Technical Library CD contains product manuals and is included with your software. The DynaText reader (included on the Technical Library CD) allows you to access technical information about your product in an easy-to-use format.

Refer to the *Technical Library Installation Guide* in your documentation package for instructions on installing and starting the Technical Library.

- The Technical Library Product Manuals Web site is an HTML version of the Technical Library CD that you can access using a standard Web browser. In addition to product manuals, you will find links to EBFs/Updates, Technical Documents, Case Management, Solved Cases, newsgroups, and the Sybase Developer Network.

To access the Technical Library Product Manuals Web site, go to Product Manuals at <http://www.sybase.com/support/manuals/>.

Sybase certifications on the Web

Technical documentation at the Sybase Web site is updated frequently.

❖ **Finding the latest information on product certifications**

- 1 Point your Web browser to Technical Documents at <http://www.sybase.com/support/techdocs/>.
- 2 Select Products from the navigation bar on the left.
- 3 Select a product name from the product list and click Go.
- 4 Select the Certification Report filter, specify a time frame, and click Go.
- 5 Click a Certification Report title to display the report.

❖ **Creating a personalized view of the Sybase Web site (including support pages)**

Set up a MySybase profile. MySybase is a free service that allows you to create a personalized view of Sybase Web pages.

- 1 Point your Web browser to Technical Documents at <http://www.sybase.com/support/techdocs/>.
- 2 Click MySybase and create a MySybase profile.

Sybase EBFs and software updates

❖ **Finding the latest information on EBFs and software updates**

- 1 Point your Web browser to the Sybase Support Page at <http://www.sybase.com/support>.
- 2 Select EBFs/Updates. Enter user name and password information, if prompted (for existing Web accounts) or create a new account (a free service).
- 3 Select a product.
- 4 Specify a time frame and click Go.
- 5 Click the Info icon to display the EBF/Update report, or click the product description to download the software.

Conventions

This section describes conventions used in this manual.

Formatting SQL statements

SQL is a free-form language. There are no rules about the number of words you can put on a line or where you must break a line. However, for readability, all examples and syntax statements in this manual are formatted so that each clause of a statement begins on a new line. Clauses that have more than one part extend to additional lines, which are indented.

Font and syntax conventions

The font and syntax conventions used in this manual are shown in Table 1.0:

Table 1: Font and syntax conventions in this manual

Element	Example
Command names, command option names, utility names, utility flags, and other keywords are bold.	select sp_configure
Database names, datatypes, file names and path names are in <i>italics</i> .	<i>master database</i>
Variables, or words that stand for values that you fill in, are in <i>italics</i> .	select <i>column_name</i> from <i>table_name</i> where <i>search_conditions</i>
Parentheses are to be typed as part of the command.	compute row_aggregate (<i>column_name</i>)
Curly braces indicate that you must choose at least one of the enclosed options. Do not type the braces.	{cash, check, credit}
Brackets mean choosing one or more of the enclosed options is optional. Do not type the brackets.	[anchovies]
The vertical bar means you may select only one of the options shown.	{die_on_your_feet live_on_your_knees live_on_your_feet}
The comma means you may choose as many of the options shown as you like, separating your choices with commas to be typed as part of the command.	[extra_cheese, avocados, sour_cream]

Element	Example
An ellipsis (...) means that you can <i>repeat</i> the last unit as many times as you like.	<pre>buy thing = price [cash check credit] [, thing = price [cash check credit]]...</pre> <p>You must buy at least one thing and give its price. You may choose a method of payment: one of the items enclosed in square brackets. You may also choose to buy additional things: as many of them as you like. For each thing you buy, give its name, its price, and (optionally) a method of payment.</p>

- Syntax statements (displaying the syntax and all options for a command) appear as follows:

```
sp_dropdevice [ device_name]
```

or, for a command with more options:

```
select column_name
      from table_name
      where search_conditions
```

In syntax statements, keywords (commands) are in normal font and identifiers are in lowercase: normal font for keywords, italics for user-supplied words.

- Examples of output from the computer appear as follows:

```
0736 New Age Books Boston MA
0877 Binnet & Hardley Washington DC
1389 Algodata Infosystems Berkeley CA
```

Case

In this manual, most of the examples are in lowercase. However, you can disregard case when typing Transact-SQL keywords. For example, `SELECT`, `Select`, and `select` are the same. Note that Adaptive Server's sensitivity to the case of database objects, such as table names, depends on the sort order installed on Adaptive Server. You can change case sensitivity for single-byte character sets by reconfiguring the Adaptive Server sort order.

See in the *System Administration Guide* for more information.

Expressions

Adaptive Server syntax statements use the following types of expressions:

Table 2: Types of expressions used in syntax statements

Usage	Definition
<i>expression</i>	Can include constants, literals, functions, column identifiers, variables, or parameters
<i>logical expression</i>	An expression that returns TRUE, FALSE, or UNKNOWN
<i>constant expression</i>	An expression that always returns the same value, such as “5+3” or “ABCDE”
<i>float_expr</i>	Any floating-point expression or expression that implicitly converts to a floating value
<i>integer_expr</i>	Any integer expression, or an expression that implicitly converts to an integer value
<i>numeric_expr</i>	Any numeric expression that returns a single value
<i>char_expr</i>	Any expression that returns a single character-type value
<i>binary_expression</i>	An expression that returns a single <i>binary</i> or <i>varbinary</i> value

Examples

Many of the examples in this manual are based on a database called pubtune. The database schema is the same as the pubs2 database, but the tables used in the examples have more rows: titles has 5000, authors has 5000, and titleauthor has 6250. Different indexes are generated to show different features for many examples, and these indexes are described in the text.

The pubtune database is not provided with Adaptive Server. Since most of the examples show the results of commands such as set showplan and set statistics io, running the queries in this manual on pubs2 tables will not produce the same I/O results, and in many cases, will not produce the same query plans as those shown here.

If you need help

Each Sybase installation that has purchased a support contract has one or more designated people who are authorized to contact Sybase Technical Support. If you cannot resolve a problem using the manuals or online help, please have the designated person contact Sybase Technical Support or the Sybase subsidiary in your area.

Introduction to Performance and Tuning

Tuning Adaptive Server Enterprise for performance can involve several processes in analyzing the “Why?” of slow performance, contention, optimizing and usage.

.This manual covers the basics for understanding and investigating performance questions in Adaptive Server. It guides you in how to look for the places that may be impeding performance.

The remaining manuals for the Performance and Tuning Series are:

Performance and Tuning: Locking

Adaptive Server locks the tables, data pages, or data rows currently used by active transactions by locking them. Locking is a concurrency control mechanism: it ensures the consistency of data within and across transactions. Locking is needed in a multiuser environment, since several users may be working with the same data at the same time.

Carefully considered indexes, built on top of a good database design, are the foundation of a high-performance Adaptive Server installation. However, adding indexes without proper analysis can reduce the overall performance of your system. Insert, update, and delete operations can take longer when a large number of indexes need to be updated.

- *Performance and Tuning: Optimizer and Abstract Plans*

The Optimizer in the Adaptive Server takes a query and finds the best way to execute it. The optimization is done based on the statistics for a database or table. The optimized plan stays in effect until the statistics are updated or the query changes. You can update the statistics on the entire table or by sampling on a percentage of the data.

Adaptive Server can generate an abstract plan for a query, and save the text and its associated abstract plan in the `sysqueryplanssystem` table. Abstract plans provide an alternative to options that must be specified in the batch or query in order to influence optimizer decisions. Using abstract plans, you can influence the optimization of a SQL statement without having to modify the statement syntax.

- *Performance and Tuning: Monitoring and Analyzing*

Adaptive Server employs reports for monitoring the server. This manual explains how statistics are obtained and used for monitoring and optimizing. The stored procedure `sp_sysmon` produces a large report that shows the performance in Adaptive Server.

You can also use the Sybase Monitor in Sybase Central for realtime information on the status of the server.

Each of the manuals has been set up to cover specific information that may be used by the system administrator and the database administrator.

Topic	Page
Good performance	3
Tuning performance	4
Identifying system limits	12
Setting tuning goals	14
Analyzing performance	14

Good performance

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

Response time

Response time is the time that a single task takes to complete. The response time can be shortened by:

- Reducing contention and wait times, particularly disk I/O wait times
- Using faster components
- Reducing the amount of time the resources are needed

In some cases, Adaptive Server is optimized to reduce initial response time, that is, the time it takes to return the first row to the user.

This is especially useful in applications where a user may retrieve several rows with a query and then browse through them slowly with a front-end tool.

Throughput

Throughput refers to the volume of work completed in a fixed time period. There are two ways of thinking of throughput:

- As a single transaction, for example, 5 UpdateTitle transactions per minute, or
- As the entire Adaptive Server, for example, 50 or 500 server-wide transactions per minute

Throughput is commonly measured in transactions per second (tps), but it can also be measured per minute, per hour, per day, and so on.

When you set the various limits for Adaptive Server it means that the server may have to handle large volumes of data for a single query, DML operation, or command. For example, if you use a data-only-locked (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

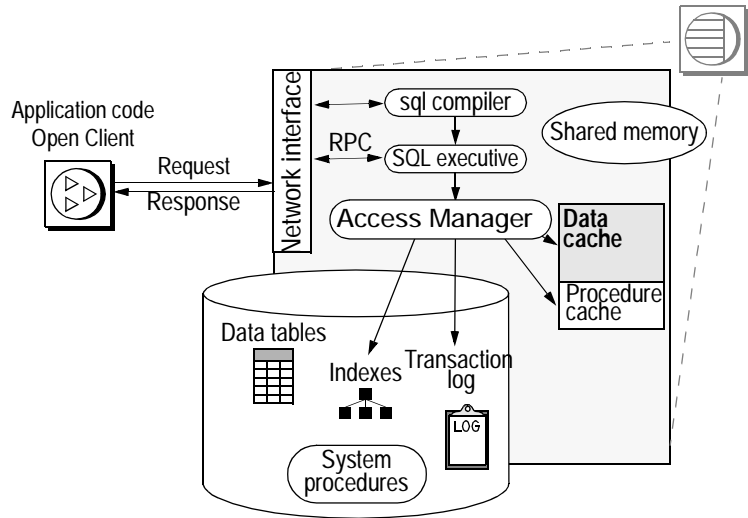
Designing for performance

Most of the gains in performance derive from good database design, thorough query analysis, and appropriate indexing. The largest performance gains can be realized by establishing a good database design and by learning to work with the Adaptive Server query optimizer as you develop your applications.

Other considerations, such as hardware and network analysis, can locate performance bottlenecks in your installation.

Tuning performance

Tuning is optimizing performance. A system model of Adaptive Server and its environment can be used to identify performance problems at each layer.

Figure 2-1: Adaptive Server system model

A major part of tuning is reducing the contention for system resources. As the number of users increases, contention for resources such as data and procedure caches, spinlocks on system resources, and the CPU(s) increases. The probability of locking data pages also increases.

Tuning levels

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

In some cases, removing a resource bottleneck at one layer can reveal another problem area. On a more optimistic note, resolving one problem can sometimes alleviate other problems.

For example, if physical I/O rates are high for queries, and you add more memory to speed response time and increase your cache hit ratio, you may ease problems with disk contention.

The following information is on the tuning layers for Adaptive Server.

Application layer

Most performance gains come from query tuning, based on good database design. This guide is devoted to an explanation of Adaptive Server internals with query processing techniques and tools to maintain high performance.

Issues at the application layer include the following:

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

Options to address these issues include:

- Using remote or replicated processing to move decision support off the OLTP machine
- Using stored procedures to reduce compilation time and network usage
- Using the minimum locking level that meets your application needs

Database layer

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

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

The "overhead" is space reserved to the server, not available for any user database. It is:

- size of the master database,
- plus size of the model database,
- plus size of tempdb
- (12.0 and beyond) plus size of sybssystemdb,
- plus 8k bytes for the server's configuration area.

Issues at the database layer include:

- Developing a backup and recovery scheme
- Distributing data across devices
- Auditing affects performance; audit only what you need
- Scheduling maintenance activities that can slow performance and lock users out of tables

Options to address these issues include:

- Using transaction log thresholds to automate log dumps and avoid running out of space
- Using thresholds for space monitoring in data segments
- Using partitions to speed loading of data
- Placing objects on devices to avoid disk contention or to take advantage of I/O parallel.
- Caching for high availability of critical tables and indexes

Adaptive Server layer

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

Issues at the Adaptive Server layer are as follows:

- The application types to be supported: OLTP, DSS, or a mix.
- The number of users to be supported can affect tuning decisions—as the number of users increases, contention for resources can shift.
- Network loads.
- Replication Server® or other distributed processing can be an issue when the number of users and transaction rate reach high levels.

Options to address these issues include:

- Tuning memory (the most critical configuration parameter) and other parameters.
- Deciding on client vs. server processing—can some processing take place at the client side?
- Configuring cache sizes and I/O sizes.

- Adding multiple CPUs.
- Scheduling batch jobs and reporting for off-hours.
- Reconfiguring certain parameters for shifting workload patterns.
- Determining whether it is possible to move DSS to another Adaptive Server.

Devices layer

This layer is for the disk and controllers that store your data. Adaptive Server can manage up to 256 devices.

Issues at the devices layer include:

- You mirror the master device, the devices that hold the user database, or the database logs?
- How do you distribute system databases, user databases, and database logs across the devices?
- Do you need partitions for parallel query performance or high insert performance on heap tables?

Options to address these issues include:

- Using more medium-sized devices and controllers may provide better I/O throughput than a few large devices
- Distributing databases, tables, and indexes to create even I/O load across devices
- Using segments and partitions for I/O performance on large tables used in parallel queries

Network layer

This layer has the network or networks that connect users to Adaptive Server.

Virtually all users of Adaptive Server access their data via the network. Major issues with the network layer are:

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

Options to address these issues include:

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

Hardware layer

This layer concerns the CPUs available.

Issues at the hardware layer include:

- CPU throughput
- Disk access: controllers as well as disks
- Disk backup
- Memory usage

Options to address these issues include:

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

Operating – system layer

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

At the operating system layer, the major issues are:

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

- CPU availability and allocation to Adaptive Server

Options include:

- Network interface
- Choosing between files and raw partitions
- Increasing the memory size
- Moving client operations and batch processing to other machines
- Multiple CPU utilization for Adaptive Server

Configuration parameters

Table 2-1 summarizes the configuration parameters.

Table 2-1: Configuration parameters

Parameter	Function
allocate max shared memory	Determines whether Adaptive Server allocates all the memory specified by max memory at start-up or only the amount of memory the configuration parameter requires.
cis bulk insert array size	Controls the size of the array when performing a bulk transfer of data from one Adaptive Server to another. During the transfer, CIS buffers rows internally, and asks the Open Client bulk library to transfer them as a block.
dynamic allocation on demand	Determines when memory is allocated for changes to dynamic memory configuration parameters.
enable enterprise java beans	Enables or disables the EJB Server.
enable file access	Enables or disables access through proxy tables to the External File System. Requires a license for ASE_XFS.
enable full-text search	Enables or disables Enhances Full-Text Search services. Requires a license for ASE_EFTS.
enable row level access control	Enables or disables row level access control.
enable ssl	Enables or disables Secure Sockets Layer session-based security
enable surrogate processing	Enables or disables the processing and maintains the integrity of surrogate pairs in Unicode data.
enable unicode normalization	Enables or disables Unilib character normalization.
heap memory per user	Specifies the heap memory per user for Adaptive Server.
max memory	Specifies the maximum amount of total logical memory that you can configure Adaptive Server to allocate.

Parameter	Function
number of engines at startup	Specifies the number of engines that are brought online at startup. This replaces the minimum online engines parameter.
number of java sockets	Specifies the maximum amount of total physical memory that you can configure Adaptive Server to allocate.
procedure cache size	Specifies the size of the procedure cache in 2K pages.
total logical memory	Specifies the amount memory that Adaptive Server is configured to use.
total physical memory	Displays the amount of memory that is being used by Adaptive Server at a given moment in time.
total memory	Displays the total logical memory for the current configuration of Adaptive Server
size of process object heap	Now a server-wide setting and not assigned to a specific task.

Dynamic

Table 2-2: Dynamic configuration parameters

Configuration parameter	Configuration parameter
addition network memory	number of pre-allocated extents
audit queue size	number of user connections
cpu grace time	number of worker processes
deadlock pipe max messages	open index hash spinlock ratio
default database size	open index spinlock ratio
default fill factor percent	open object spinlock ratio
disk i/o structures	partition groups
errorlog pipe max messages	partition spinlock ratio
max cis remote connections	permission cache entries
memory per worker process	plan text pipe max messages
number of alarms	print recovery information
number of aux scan descriptors	process wait events
number of devices	size of global fixed heap
number of dtx participants	size of process object heap
number of java sockets	size of shared class heap
number of large i/o buffers	size of unilib cache
number of locks	sql text pipe max messages
number of mailboxes	statement pipe max messages
number of messages	tape retention in days
number of open databases	time slice
number of open indexes	user log cache spinlock ratio

Configuration parameter	Configuration parameter
number of open objects	

Identifying system limits

There are limits to maximum performance. The physical limits of the CPU, disk subsystems, and networks impose limits. Some of these can be overcome by adding memory, using faster disk drives, switching to higher bandwidth networks, and adding CPUs.

Given a set of components, any individual query has a minimum response time. Given a set of system limitations, the physical subsystems impose saturation points.

Varying logical page sizes

Adaptive Server version 12.5 does 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.

Number of columns and column size

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

- 1024 for fixed-length columns in both all-pages-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 any 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.

Maximum length of expressions, variables, and stored procedure arguments

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

Number of logins

Table 2-3 lists the limits for the number of logins, users, and groups for Adaptive Server.

Table 2-3: Limits for number of logins, users, and groups

Item	Version 12.0 limit	Version 12.5 limit	New range
Number of logins per server (SUID)	64K	2 billion plus 32K	-32768 to 2 billion
Number of users per database	48K	2 billion less 1032193	-32768 to 16383; 1048577 to 2 Billion
Number of groups per database	16K	1032193	16384 to 1048576

Performance implications for limits

The limits set for Adaptive Server mean that the server may have to handle large volumes of data for a single query, DML operation, or command. For example, if you use a data-only-locked (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

Setting tuning goals

For many systems, a performance specification developed early in the application life cycle sets out the expected response time for specific types of queries and the expected throughput for the system as a whole.

Analyzing performance

When there are performance problems, you need to determine the sources of the problems and your goals in resolving them. The steps for analyzing performance problems are:

- 1 Collect performance data to get baseline measurements. For example, you might use one or more of the following tools:
 - Benchmark tests developed in-house or industry-standard third-party tests.
 - `sp_sysmon`, a system procedure that monitors Adaptive Server performance and provides statistical output describing the behavior of your Adaptive Server system.
See [Performance and Tuning Guide: Monitoring and Analyzing for Performance](#) for information on using `sp_sysmon`.
 - Adaptive Server Monitor provides graphical performance and tuning tools and object-level information on I/O and locks.
 - Any other appropriate tools.

- 2 Analyze the data to understand the system and any performance problems. Create and answer a list of questions to analyze your Adaptive Server environment. The list might include questions such as:
 - What are the symptoms of the problem?
 - What components of the system model affect the problem?
 - Does the problem affect all users or only users of certain applications?
 - Is the problem intermittent or constant?
- 3 Define system requirements and performance goals:
 - How often is this query executed?
 - What response time is required?
- 4 Define the Adaptive Server environment—know the configuration and limitations at all layers.
- 5 Analyze application design – examine tables, indexes, and transactions.
- 6 Formulate a hypothesis about possible causes of the performance problem and possible solutions, based on performance data.
- 7 Test the hypothesis by implementing the solutions from the last step:
 - Adjust configuration parameters.
 - Redesign tables.
 - Add or redistribute memory resources.
- 8 Use the same tests used to collect baseline data in step 1 to determine the effects of tuning. Performance tuning is usually a repetitive process.

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

Normal Forms

Usually, several techniques are used to reorganize a database to minimize and avoid inconsistency and redundancy, such as Normal Forms.

Using the different levels of Normal Forms organizes the information in such a way that it promotes efficient maintenance, storage and updating. It simplifies query and update management, including the security and integrity of the database. However, such normalization usually creates a larger number of tables which may in turn increase the size of the database.

Database Administrators must decide the various techniques best suited their environment.

Use the *Adaptive Server Reference Manual* as a guide in setting up databases.

Locking

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

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

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

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

Locking for performance is discussed in the *Performance and Tuning: Locking*, manual see the chapters on Configuring and Tuning, Using Locking Commands and Reports on Locking.

Special Considerations

Databases are allocated among the devices in fragments called "disk pieces", where each disk piece is represented by one entry in master.dbo.sysusages. Each disk piece:

- Represents a contiguous fragment of one device, up to the size of the device.

- Is an even multiple of 256 logical pages.

One device may be divided among many different databases. Many fragments of one device may be apportioned to one single database as different disk pieces.

There is no practical limit on the number of disk pieces in one database, except that the Adaptive Server's configured memory must be large enough to accommodate its in-memory representation.

Because disk pieces are multiples of 256 logical pages, portions of odd-sized devices may remain unused. For example, if a device has 83 Mb and the server uses a 16k page size, 256 logical pages is $256 * 16k = 4 \text{ Mb}$. The final 3 Mb of that device will not be used by any database because it's too small to make a group of 256 logical pages.

The master device sets aside its first 8k bytes as a configuration area. Thus, to avoid any wasted space, a correctly-sized master device should be an even number of 256 logical pages *plus* 8 kb.

This chapter discusses the role that the network plays in performance of applications using Adaptive Server.

Topic	Page
Introduction	19
Potential performance problems	19
How Adaptive Server uses the network	21
Changing network packet sizes	27
Impact of other server activities	31
Improving network performance	32

Introduction

Usually, the System Administrator is the first to recognize a problem on the network or in performance, including such things as:

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

Potential performance problems

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

- Adaptive Server uses network services poorly.

- The physical limits of the network have been reached.
- Processes are retrieving unnecessary data values, increasing network traffic unnecessarily.
- Processes are opening and closing connections too often, increasing network load.
- Processes are frequently submitting the same SQL transaction, causing excessive and redundant network traffic.
- Adaptive Server does not have enough network memory.
- Adaptive Server's network packet sizes are not big enough to handle the type of processing needed by certain clients.

Basic questions on network performance

When looking at problems that you think might be network-related, ask yourself these questions:

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

Techniques summary

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

- Using small packets for most database activity
- Using larger packet sizes for tasks that perform large data transfers
- Using stored procedures to reduce overall traffic
- Filtering data to avoid large transfers

- Isolating heavy network users from ordinary users
- Using client control mechanisms for special cases

Using *sp_sysmon* while changing network configuration

Use *sp_sysmon* while making network configuration changes to observe the effects on performance. Use Adaptive Server Monitor to pinpoint network contention on a particular database object.

For more information about using *sp_sysmon*, see Chapter 8, “Monitoring Performance with *sp_sysmon*,” in *Performance and Tuning Guide: Monitoring and Analyzing*.

How Adaptive Server uses the network

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

Adaptive Server was one of the first database systems to be built on a network-based client/server architecture. Clients initiate a connection to the server. The connection sends client requests and server responses. Applications can have as many connections open concurrently as they need to perform the required task.

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

Managing Network Listeners

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

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

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

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

SYBSRV1

```
master tli tcp /dev/tcp \x00020abc1234567800000000000000000000
query tli tcp /dev/tcp \x00020abc1234567800000000000000000000
master tli tcp /dev/tcp \x00020abd1234567800000000000000000000
```

This interfaces file entry includes two listener ports. For more information about the interfaces file, see [Connecting to Adaptive Server in the System Administration Guide](#).

Network Listeners on UNIX

Network listeners run on UNIX slightly differently than they do on Windows NT because on UNIX each Adaptive Server engine is a separate process, but on Windows NT, Adaptive Server is a single process.

The following are true of network listeners on UNIX:

- Adaptive Server uses one listener task per port. Each listener task functions as multiple logical listeners by switching from engine to engine, attempting to balance the load. For example, a 64-engine Adaptive Server with two master ports has two listener tasks, but these two listener tasks act as 128 logical listener tasks, so the server has two physical and 128 logical listeners. Starting a listener on engine 3 does not result in Adaptive Server spawning a new listener task unless the port does not already have a listener
- A listener task accepts connections on engines on which it is enabled. So a single listener task corresponds to many logical listeners. On Windows NT, logical listeners and listener tasks are a one to one correspondence.

- Stopping a listener on a specific engine terminates the logical listener for this engine since the listener task no longer switches to that engine. Adaptive Server terminates the listener task in case this was the last engine on which it was allowed to operate.

Managing listeners with *sp_listener*

You can manage listeners with the *sp_listener* system procedure. *sp_listener* allows you to:

- Start additional listeners (the maximum number of listeners is 32)
- Stop listeners
- Suspend listeners
- Resume suspended listeners

The syntax for *sp_listener* is:

```
sp_listener "command", "server_name", engine | remaining
```

or

```
sp_listener "command", "[protocol:]machine:port", engine | remaining
```

The maximum number of listeners you can add in addition to the listeners created at startup is 32. The semantics for *sp_listener* is atomic: if a command cannot be completed successfully, it is aborted.

Where *command* is can be start, stop, suspend, resume, or status, *server_name* is the name of Adaptive Server, *engine* specifies the number of the engine affected by this command (this parameter is ignored by Windows NT, *engine* can be a single-engine number in quotes ("2"), a list ("3,5,6"), a range ("2-5"), or mix of all ("2,3-5,7")), remaining specifies that the command is to take effect on all engines on which it can be meaningfully applied (that is, where the listener is in a state in which the command is can take effect), *protocol* is the protocol used (tcp, tli, ssltcp, ssltli, winsock, sslnlwsock, or sslwinsock), and *machine:port* is the machine name and port number (as specified in the interfaces file) to which the listener listens.

The first syntax description above is intended for all master ports listed in the interfaces file. This syntax allows you to start, stop, suspend, or resume activity simultaneously on all *master* entries under the server name in the interfaces file. The second syntax description allows you to manage listeners not listed in the interfaces file.

Both syntaxes are dynamic, that is you do not have to restart Adaptive Server to implement the change.

Note Stopping a listener that is listed in the interfaces file does not remove this entry from the interfaces file.

The examples in this chapter use an Adaptive Server named “ASE1251” running on a host named “spartacus.” The examples using the first syntax for `sp_listener` apply only to the master ports registered in the interfaces file under Adaptive Server ASE1251. Commands using the second syntax for `sp_listener` (in the form `tcp:spartacus:4556`) apply only to the single master port specified (4556 in this example). A master port is unambiguously determined by a network protocol, a hostname and a port number.

Using the *remaining* parameter

The remaining parameter specifies that, for the command you are running (start, stop, resume, and so on), the command runs successfully for all listeners that are in a state that allow the change (for example, from start to stop). For example, if you attempt to start listeners on engines one through six, but engines one, four, and five are unavailable, `sp_listener...remaining` starts listeners on engines two, three, and six, disregarding the offline engines.

Without the remaining parameter, an `sp_listener` command fails if the entire command cannot succeed for all listeners. If you do not include the remaining parameter in the example above, the command fails even though it could have started listeners on engines two, three, and six.

Determining the status of listeners

`sp_listener...status` reports on the state of the listeners. The state is one of active, stopped, or suspended. You can query about a specific listener by indicating the machine, port, or engine number, or you can query about the status of all listeners by not including any parameters.

The following queries the status of all listeners:

```
sp_listener "status"
```

The following queries the status of those listeners on engine three of Adaptive Server ASE1251 which are registered in the interfaces file:

```
sp_listener "status", ASE1251, "3"
```

The following queries the status of the tli listener on port 4556 of the current Adaptive Server running on machine spartacus:

```
sp_listener "status" "tli:spartacus:4556"
```

Starting new listeners

`sp_listener...start` starts additional listeners. These are listeners in addition to those listed in the interfaces file. You can specify that the listener start on a specific range of engines.

The command will not fail when you use the remaining parameter, in case the listener is already enabled on some engines. If you explicitly include an engine list, and the listener is running on one of the specified engines, the command fails.

For example, the following specifies that a listener starts for server ASE1251 on engine number three:

```
sp_listener "start", ASE1251, "3"
```

Or you can specify that the listener start on a certain machine and port number using a certain protocol. This example specifies that a listener start on port 4556 using the tli protocol on the IP address of machine spartacus on all engines for which this listener is not already running:

```
sp_listener "start", "tli:ASE1251:4556"
```

You can also specify a range of engine numbers. For example, the following specifies that listeners start on engines three through six, corresponding to all master ports registered in the interfaces file under server name ASE1251:

```
sp_listener "start", "ASE1251", "3-6"
```

The following starts listeners corresponding to master ports registered under server name ASE1251 in the interfaces file on all engines on which the corresponding listener is not already active:

```
sp_listener "start", "ASE1251", "3-6", "remaining"
```

Stopping listeners

The stop command terminates the specified listeners. If you specify non-existent listeners in the syntax, `sp_listener` fails without affecting the other listeners. `sp_listener...stop` also fails if you are trying to stop the last active listener on the server.

Note `sp_listener` does not run if you are attempt to stop all active listeners with the stop parameter.

The following command stops the listener corresponding to the specified tli address for all engines on which it is active:

```
sp_listener "stop", "tli:ASE1251:4556"
```

This command stops all listeners registered in the interfaces file for server ASE1251 for the specified range of engines:

```
sp_listener "stop", "ASE1251", "3-6"
```

(Windows NT only) To stop all listeners on engines three through six that are in a state that will allow the change:

```
sp_listener "stop", "ASE1251", "remaining"
```

Suspending listeners

The suspend parameter prevents the listener from accepting any more connections. `sp_listener...suspend` is less drastic than `sp_listener...stop` because it does not close the listener port on the given engine. It only informs the listener to stop accepting connections on the given engine until further notice. `sp_listener...suspend` is helpful for temporarily preventing a listener from accepting connections. The listener can resume listening with the resume parameter. `sp_listener...suspend` fails if it is suspending the last active listener on the system.

`sp_listener...suspend` only affects future connections; it does not affect the connections that are active.

To suspend a listener on engine three of Adaptive Server ASE1251:

```
sp_listener "suspend", ASE1251, 3
```

The following suspends the listener corresponding to the specified tli address for all engines on which it is not already running:


```
sp_listener "suspend", "tli:ASE1251:4556"
```

To suspend all listeners on engines three through six that are in a state that will allow the change (assuming the server has seven engines):

```
sp_listener "suspend", "ASE1251", "remaining"
```

Resume suspended listeners

`sp_listener...resume` instructs suspended listeners to resume listening and to accept new connections. For example, the following reactivates the listener on engine three of Adaptive Server ASE1251, above:

```
sp_listener "resume", ASE1251, 3
```

To reactivate listeners on port 4556 of Adaptive Server ASE1251:

```
sp_listener "resume", "tli:ASE1251:4556"
```

To reactivate all listeners on engines three through six that are in a state that will allow the change:

```
sp_listener "resume", "ASE1251", "remaining"
```

Changing network packet sizes

By default, all connections to Adaptive Server use a default packet size of 512 bytes. This works well for clients sending short queries and receiving small result sets. However, some applications may benefit from an increased packet size.

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

Decision support applications often include large batches of Transact-SQL and return larger result sets.

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

The *System Administration Guide* describes how to change these configuration parameters:

- The default network packet size, if most of your applications are performing large reads and writes
- The max network packet size and additional network memory, which provides additional memory space for large packet connections

Only a System Administrator can change these configuration parameters.

Large versus default packet sizes for user connections

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

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

Number of packets is important

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

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

- Bulk copy
- readtext and writetext commands
- select statements with large result sets

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

The results can be significantly different between applications. Bulk copy might work best at one packet size, while large image data retrievals might perform better at a different packet size.

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

Evaluation tools with Adaptive Server

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

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

You can also use these global variables:

- `@@pack_sent` – Number of packets sent by Adaptive Server
- `@@pack_received` – Number of packets received
- `@@packet_errors` – Number of errors

These SQL statements show how the counters can be used:

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

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

See *Performance and Tuning Guide: Monitoring and Analyzing for Performance* for more information about `sp_monitor` and these global variables.

Evaluation tools outside of Adaptive Server

Operating system commands also provide information about packet transfers. See the documentation for your operating system for more information about these commands.

Server-based techniques for reducing network traffic

Using stored procedures, views, and triggers can reduce network traffic. These Transact-SQL tools can store large chunks of code on the server so that only short commands need to be sent across the network. If your applications send large batches of Transact-SQL commands to Adaptive Server, converting them to use stored procedures can reduce network traffic.

- Stored procedures

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

You may be able to reduce network overhead by turning off “doneinproc” packets.

See *Performance and Tuning: Monitoring and Analyzing for Performance* for more information.

- Ask for only the information you need

Applications should request only the rows and columns they need, filtering as much data as possible at the server to reduce the number of packets that need to be sent. In many cases, this can also reduce the disk I/O load.

- Large transfers

Large transfers simultaneously decrease overall throughput and increase the average response time. If possible, large transfers should be done during off-hours. If large transfers are common, consider acquiring network hardware that is suitable for such transfers. Table 3-1 shows the characteristics of some network types.

Table 3-1: Network options

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

- Network overload

Overloaded networks are becoming increasingly common as more and more computers, printers, and peripherals are network equipped. Network managers rarely detect problems before database users start complaining to their System Administrator

Be prepared to provide local network managers with your predicted or actual network requirements when they are considering the adding resources. You should also keep an eye on the network and try to anticipate problems that result from newly added equipment or application requirements.

Impact of other server activities

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

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

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

Single user versus multiple users

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

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

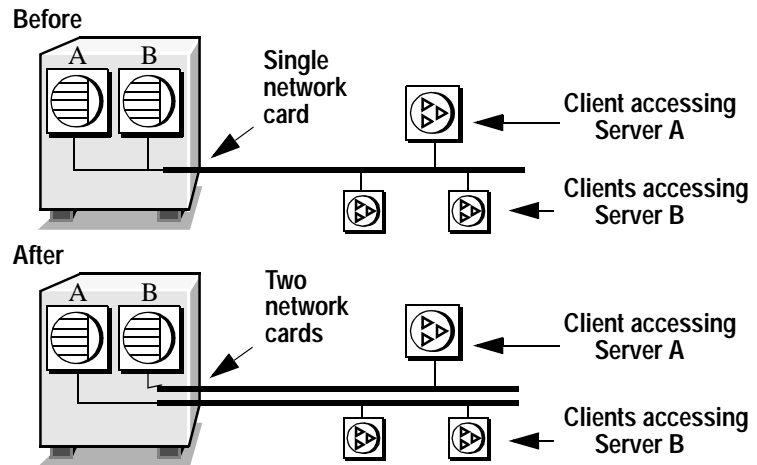
When response time is “abnormally” high, and normal tests indicate no problem, it could be due to other users on the same network. In such cases, ask the user when the process was being run, if the operating system was generally sluggish, if other users were doing large transfers, and so on.

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

Improving network performance

Isolate heavy network users

Isolate heavy network users from ordinary network users by placing them on a separate network, as shown in Figure 3-1.

Figure 3-1: Isolating heavy network users

In the “Before” diagram, clients accessing two different Adaptive Servers use one network card. Clients accessing Servers A and B have to compete over the network and past the network card.

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

Set *tcp no delay* on TCP networks

By default, the configuration parameter `tcp no delay` is set to “off,” meaning that the network performs packet batching. It briefly delays sending partial packets over the network.

While this improves network performance in terminal-emulation environments, it can slow performance for Adaptive Server applications that send and receive small batches. To disable packet batching, a System Administrator can set the `tcp no delay` configuration parameter to 1.

Configure multiple network listeners

Use two (or more) ports listening for a single Adaptive Server. Front-end software may be directed to any configured network ports by setting the DSQUERY environment variable.

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

See the Adaptive Server configuration guide for your platform for information on configuring multiple network listeners.

Using Engines and CPUs

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

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

Topic	Page
Background concepts	35
Single-CPU process model	38
Adaptive Server SMP process model	43
Asynchronous log service	47
Housekeeper task improves CPU utilization	50
Measuring CPU usage	53
Enabling engine-to-CPU affinity	55
Multiprocessor application design guidelines	57

Background concepts

This section provides an overview of how Adaptive Server processes client requests. It also reviews threading and other related fundamentals.

Like an operating - system, a relational database must be able to respond to the requests of many concurrent users. Adaptive Server is based on a multithreaded, single-process architecture that allows it to manage thousands of client connections and multiple concurrent client requests without overburdening the operating - system.

In a system with multiple CPUs, you can enhance performance by configuring Adaptive Server to run using multiple Adaptive Server *engines*. Each engine is a single operating - system process that yields high performance when you configure one engine per CPU.

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

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

How Adaptive Server processes client requests

Adaptive Server creates a new *client task* for every new connection. It fulfills a client request as outlined in the following steps:

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

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

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

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

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

Client task implementation

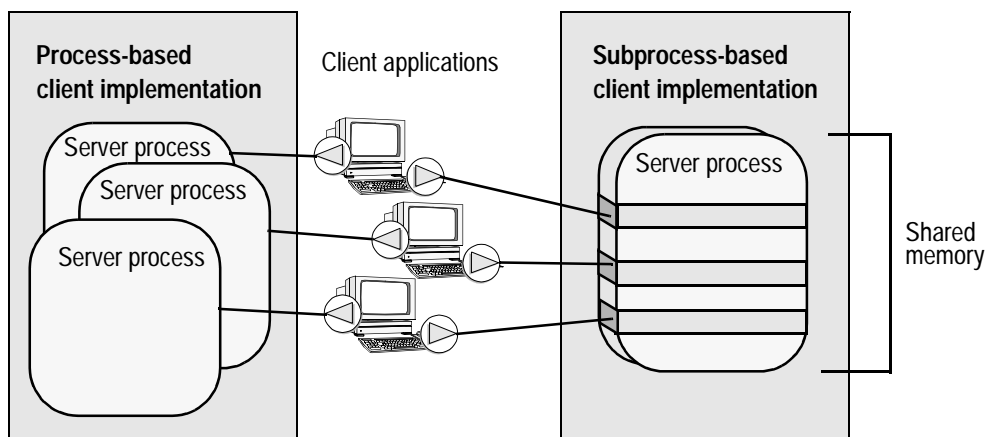
Adaptive Server client tasks are implemented as subprocesses, or “lightweight processes,” instead of operating - system processes, as subprocesses use only a small fraction of the resources that processes use.

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

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

Figure 4-1 illustrates the difference in system resources required by client connections implemented as processes and client connections implemented as subprocesses. Subprocesses exist and operate within a single instance of the executing program process and its address space in shared memory.

Figure 4-1: Process versus subprocess architecture



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

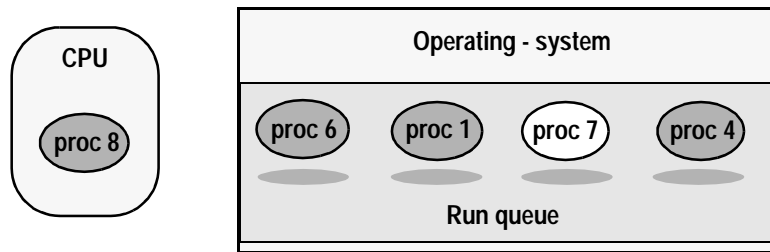
Single-CPU process model

In a single-CPU system, Adaptive Server runs as a single process, sharing CPU time with other processes, as scheduled by the operating - system. This section is an overview of how an Adaptive Server system with a single CPU uses the CPU to process client requests.

“Adaptive Server SMP process model” on page 43 expands on this discussion to show how an Adaptive Server system with multiple CPUs processes client requests.

Scheduling engines to the CPU

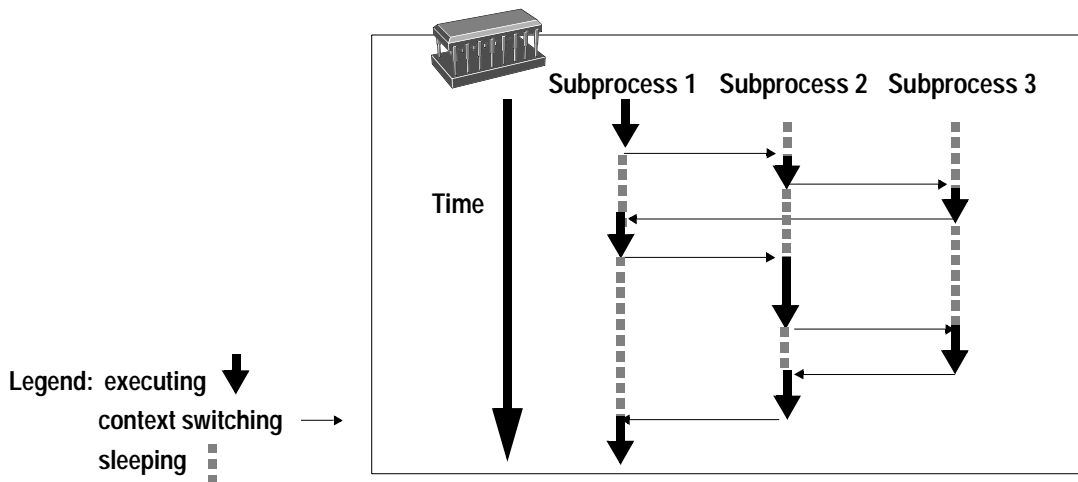
Figure 4-2 shows a run queue for a single-CPU environment in which process 8 (proc 8) is running on the CPU and processes 6, 1, 7, and 4 are in the operating - system run queue waiting for CPU time. Process 7 is an Adaptive Server process; the others can be any operating - system process.

Figure 4-2: Processes queued in the run queue for a single CPU

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

Figure 4-3 shows three subprocesses in a multitasking environment. The subprocesses are represented by the thick, dark arrows pointing down. The subprocesses share a single CPU by switching onto and off the engine over time. They are using CPU time when they are solid – near the arrowhead. They are in the run queue waiting to execute or sleeping while waiting for resources when they are represented by broken lines.

Note that, at any one time, only one process is executing. The others sleep in various stages of progress.

Figure 4-3: Multithreaded processing

Scheduling tasks to the engine

Figure 4-4 shows tasks (or worker processes) queued up for an Adaptive Server engine in a single-CPU environment. This figure switches from Adaptive Server in the operating - system context (as shown in Figure 4-2 on page 39) to Adaptive Server internal task processing. Adaptive Server, not the operating - system, dynamically schedules client tasks from the run queue onto the engine. When the engine finishes processing one task, it executes the task at the head of the run queue.

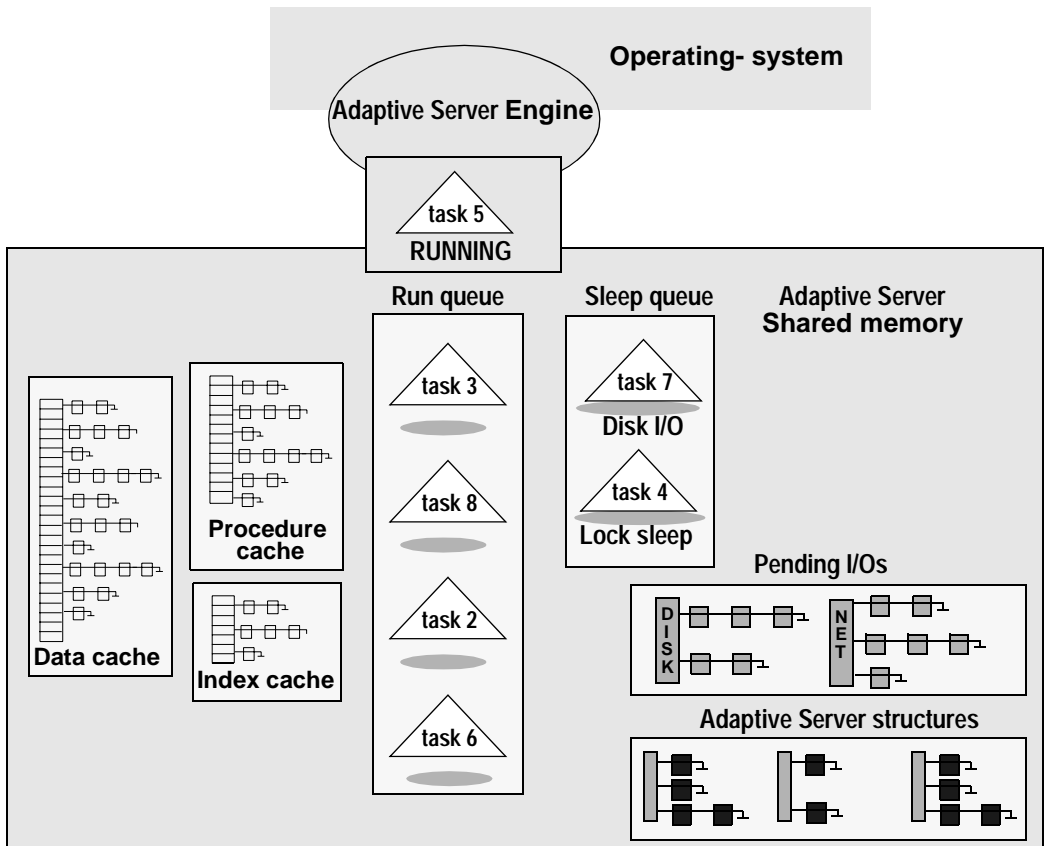
After a task begins running on the engine, the engine continues processing it until one of the following events occurs:

- The task needs a resource such as a page that is locked by another task, or it needs to perform a slow job such as disk I/O or network I/O. The task is put to sleep, waiting for the resource.
- The task runs for a configurable period of time and reaches a yield point. Then the task relinquishes the engine, and the next process in the queue starts to run. “Scheduling client task processing time” on page 42 discusses in more detail how this works.

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

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

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



Execution task scheduling

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

Scheduling client task processing time

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

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

Each time the task comes to a yield point, it checks to see if time slice has been exceeded. If it has not, the task continues to execute. If execution time does exceed time slice, the task voluntarily relinquishes the engine within the cpu grace time interval and the next task in the run queue begins executing.

The default value for the time slice parameter is 100 clock milliseconds, and there is seldom any reason to change it. The default value for cpu grace time is 500 clock ticks. If time slice is set too low, an engine may spend too much time switching between tasks, which tends to increase response time.

If time slice is set too high, CPU-intensive processes may monopolize the CPU, which can increase response time for short tasks. If your applications encounter time slice errors, adjust cpu grace time, **not** time slice.

See Chapter 5, "Distributing Engine Resources," for more information.

Use `sp_sysmon` to determine how many times tasks yield voluntarily.

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

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

If no other tasks are in the run queue, and the engine still has CPU time, Adaptive Server continues to grant time slice intervals to the task until it completes.

Normally, tasks relinquish the engine at yield points prior to completion of the cpu grace time interval. It is possible for a task not to encounter a yield point and to exceed the time slice interval. When the cpu grace time ends, Adaptive Server terminates the task with a time slice error. If you receive a time slice error, try increasing the time up to four times the current time for cpu grace time. If the problem persists, call Sybase Technical Support.

Maintaining CPU availability during idle time

When Adaptive Server has no tasks to run, it loops (holds the CPU), looking for executable tasks. The configuration parameter `runnable process search count` controls the number of times that Adaptive Server loops.

With the default value of 2000, Adaptive Server loops 2000 times, looking for incoming client requests, completed disk I/Os, and new tasks in the run queue. If there is no activity for the duration of `runnable process search count`, Adaptive Server relinquishes the CPU to the operating - system.

Note If you are having performance problems, try setting `runnable process search count` to 3.

The default for `runnable process search count` generally provides good response time, if the operating - system is not running clients other than Adaptive Server.

Use `sp_sysmon` to determine how `runnable process search count` affects Adaptive Server's use of CPU cycles, engine yields to the operating - system, and blocking network checks.

See *Performance and Tuning Guide: Monitoring and Analyzing for Performance* on using the `sp_sysmon`.

Adaptive Server SMP process model

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

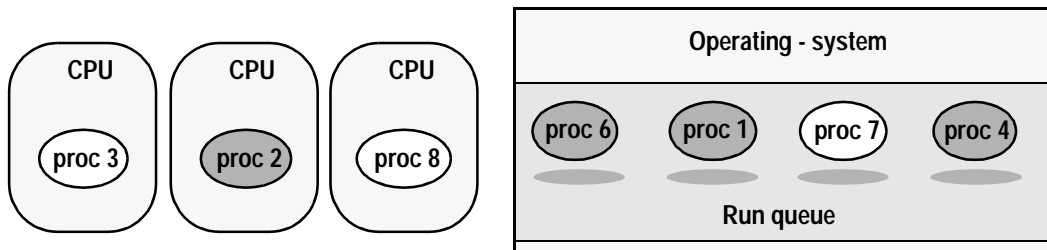
SMP is intended for machines with the following features:

- A symmetric multiprocessing operating - system
- Shared memory over a common bus
- Two to 128 processors
- Very high throughput

Scheduling engines to CPUs

In a system with multiple CPUs, multiple processes can run concurrently. Figure 4-5 represents Adaptive Server engines as the nonshaded ovals waiting in the operating - system run queue for processing time on one of three CPUs. It shows two Adaptive Server engines, proc 3 and proc 8, being processed simultaneously.

Figure 4-5: Processes queued in the OS run queue for multiple CPUs



The *symmetric* aspect of SMP is a lack of affinity between processes and CPUs—processes are not attached to a specific CPU. Without CPU affinity, the operating - system schedules engines to CPUs in the same way as it schedules non-Adaptive Server processes to CPUs. If an Adaptive Server engine does not find any runnable tasks, it can either relinquish the CPU to the operating - system or continue to look for a task to run by looping for the number of times set in the runnable process search count configuration parameter.

Scheduling Adaptive Server tasks to engines

Scheduling Adaptive Server tasks to engines in the SMP environment is similar to scheduling tasks in the single-CPU environment, as described in “Scheduling tasks to the engine” on page 40. The difference is that in the SMP environment:

- Each engine has a run queue. Tasks have soft affinities to engines. When a task runs on an engine, it creates an affinity to the engine. If a task yields the engine and then is queued again, it tends to be queued on the same engine's run queue.
- Any engine can process the tasks in the global run queue (unless logical process management has been used to assign the task to a particular engine or set of engines).

Multiple network engines

Each Adaptive Server engine handles the network I/O for its connections. Engines are numbered sequentially, starting with engine 0.

When a user logs in to Adaptive Server, the task is assigned in round-robin fashion to one of the engines that will serve as its *network engine*. This engine handles the login to establish packet size, language, character set, and other login settings. All network I/O for a task is managed by its network engine until the task logs out.

Task priorities and run queues

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

See Chapter 5, “Distributing Engine Resources,” for more information on performance tuning and task priorities.

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

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

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

Processing scenario

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

1 Assigning a network engine during login

When a connection logs in to Adaptive Server, it is assigned to an engine that will manage its network I/O. This engine then handles the login.

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

2 Checking for client requests

Another engine checks for incoming client requests once every clock tick.

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

3 Fulfilling a client request

When a task becomes first in the queue, the engine parses, compiles, and begins executing the steps defined in the task’s query plan

4 Performing disk I/O

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

5 Performing network I/O

When the task needs to return results to the user, the engine on which it is executing issues the network I/O request, and puts the tasks to sleep on a network write.

The engine checks once each clock tick to determine whether the network I/O has completed. When the I/O has completed, the task is placed on the run queue for the engine to which it is affiliated, or the global run queue.

Asynchronous log service

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

You cannot use ALS if you have fewer than 4 engines. If you try to enable ALS with fewer than 4 online engines an error message appears.

Enabling ALS

You can enable, disable, or configure ALS using the `sp_dboption` stored procedure.

```
sp_dboption <db Name>, "async log service",
"true|false"
```

Issuing a checkpoint

After issuing `sp_dboption`, you must issue a checkpoint in the database for which you are setting the ALS option:

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

You can use the checkpoint to identify the one or more databasess or use an all clause.

```
checkpoint [all | [dbname[, dbname[, dbname....]]]
```

Disabling ALS

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

```
sp_dboption "mydb", "async log service", "false"
use mydb
checkpoint
-----
```

```
Error 3647: Cannot put database in single-user mode.
Wait until all users have logged out of the database and
issue a CHECKPOINT to disable "async log service".
```

If there are no active users in the database, this example disables ALS:

```
sp_dboption "mydb", "async log service", "false"
use mydb
checkpoint
-----
```

Displaying ALS

You can see whether ALS is enabled in a specified database by checking `sp_helpdb`.

```
sp_helpdb "mydb"
-----
mydb                3.0 MB sa                2
```

July 09, 2002

```
select into/bulkcopy/pllsort, trunc log on chkpt,  
      async log service
```

Understanding the user log cache (ULC) architecture

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

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

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

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

- 4 Flush the log cache to disk.

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

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

When to use ALS

You can enable ALS on any specified database that has at least one of the following performance issues, so long as your systems runs 4 or more online engines:

- Heavy contention on the last log page.

You can tell that the last log page is under contention when the `sp_sysmon` output in the Task Management Report section shows a significantly high value. For example:

Table 4-1: Log page under contention

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

- Heavy contention on the cache manager spinlock for the log cache.

You can tell that the cache manager spinlock is under contention when the `sp_sysmon` output in the Data Cache Management Report section for the database transaction log cache shows a high value in the Spinlock Contention section. For example:

Table 4-2:

Cache c_log	per sec	per xact	count	% of total
Spinlock Contention	n/a	n/a	n/a	40.0%

- Underutilized bandwidth in the log device.

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

Using the ALS

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

- The User Log Cache (ULC) flusher
- The Log Writer.

ULC flusher

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

Log writer

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

Changes in stored procedures

Asynchronous log service changes the stored procedures `sp_dboption` and `sp_helpdb`:

- `sp_dboption` adds an option that enables and disables ALS.
- `sp_helpdb` adds a column to display ALS.

For more information on `sp_helpdb` and `sp_dboption`, see the *Reference Manual*.

Housekeeper task improves CPU utilization

When Adaptive Server has no user tasks to process, the housekeeper wash task and the housekeeper chores task automatically begin writing dirty buffers to disk and performing other maintenance tasks. These writes are done only by the housekeeper wash task during the server's idle cycles, and are known as *free writes*. They result in improved CPU utilization and a decreased need for buffer washing during transaction processing. They also reduce the number and duration of checkpoint spikes (times when the checkpoint process causes a short, sharp rise in disk writes).

Another housekeeper task is housekeeper garbage collection, which operates at the priority level of the ordinary user. It cleans up data that was logically deleted and resets the rows so the tables have space again.

Side effects of the housekeeper task

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

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

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

Configuring the housekeeper task

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

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

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

See Monitoring Performance with sp_sysmon in the *Performance and Tuning: Monitoring and Analyzing for Performance* manual.

Changing the percentage by which writes can be increased

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

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

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

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

Disabling the housekeeper task

You may want to disable the housekeeper wash and the housekeeper chores task to establish a controlled environment in which only specified user tasks are running. To disable these housekeeper tasks, set the value of the housekeeper free write percent parameter to 0:

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

Warning! In addition to buffer washing, the housekeeper periodically flushes statistics to system tables. These statistics are used for query optimization, and incorrect statistics can severely reduce query performance. Do not set the housekeeper free write percent to 0 on a system where data modification commands may be affecting the number of rows and pages in tables and indexes.

Allowing the housekeeper task to work continuously

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

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

The “Recovery management” on page 99 in the *Performance and Tuning: Monitoring and Analyzing for Performance* manual section of `sp_sysmon` shows checkpoint information to help you determine the effectiveness of the housekeeper.

Measuring CPU usage

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

Single-CPU machines

There is no correspondence between your operating - system's reports on CPU usage and Adaptive Server's internal "CPU busy" information. It is normal for an Adaptive Server to exhibit very high CPU usage while performing an I/O-bound task.

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

Using *sp_monitor* to measure CPU usage

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

last_run	current_run	seconds	
Jul 28 1999 5:25PM	Jul 28 1999 5:31PM	360	
cpu_busy	io_busy	idle	
5531(359)-99%	0(0)-0%	178302(0)-0%	
packets_received	packets_sent	packet_errors	
57650(3599)	60893(7252)	0(0)	
total_read	total_write	total_errors	connections
190284(14095)	160023(6396)	0(0)	178(1)

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

Using `sp_sysmon` to measure CPU usage

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

The “CPU Yields by engine” section displays information about how often the engine yielded to the operating - system during the interval.

See Monitoring Performance with `sp_sysmon` in the *Performance and Tuning: Monitoring and Analyzing* book for more information about `sp_sysmon`.

Operating - system commands and CPU usage

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

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

Determining when to configure additional engines

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

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

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

Before configuring more engines, use `sp_sysmon` to establish a baseline. Look at the `sp_sysmon` output for the following sections in Monitoring Performance with `sp_sysmon` in the *Performance and Tuning: Monitoring and Analyzing* manual.

In particular, study the lines or sections in the output that may reveal points of contention in the book *Performance and Tuning: Monitoring and Analyzing for Performance*:

- “Logical lock contention” on page 32.
- “Address lock contention” on page 33.
- “ULC semaphore requests” on page 57.
- “Log semaphore requests” on page 58.
- “Page splits” on page 63.
- “Lock summary” on page 76.
- “Cache spinlock contention” on page 89.
- “I/Os delayed by” on page 103.

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

Taking engines offline

`dbcc (engine)` can be used to take engines offline. The syntax is:

```
dbcc engine(offline, [enginenum])
```

```
dbcc engine(“online”)
```

If *enginenum* is not specified, the highest-numbered engine is taken offline. For more information, see the *System Administration Guide*.

Enabling engine-to-CPU affinity

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

Not all operating - systems support CPU affinity. The `dbcc tune` command is silently ignored on systems that do not support engine-to-CPU affinity. The `dbcc tune` command must be reissued each time Adaptive Server is restarted. Each time CPU affinity is turned on or off, Adaptive Server prints a message in the error log indicating the engine and CPU numbers affected:

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

The syntax is:

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

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

$$((start_cpu + n) \% number_of_cpus)$$

CPU numbers range from 0 through the number of CPUs minus 1.

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

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

The command gives this result:

Engine	CPU
0	2 (the <code>start_cpu</code> number specified)
1	3
2	0
3	1

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

Engine	CPU
0	2
1	3
2	0

In this example, CPU 1 is not used by Adaptive Server.

To disable CPU affinity, use -1 in place of `start_cpu`, and specify `off` for the setting:

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

You can enable CPU affinity without changing the value of `start_cpu` by using `-1` and `on` for the setting:

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

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

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

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

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

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

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

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

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

Multiprocessor application design guidelines

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

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

- Multiple indexes

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

For information about the effects of index maintenance on performance, see “Index management” on page 60 in the *Performance and Tuning: Monitoring and Analyzing for Performance* book.

- Managing disks

The additional processing power of SMP may increase demands on the disks. Therefore, it is best to spread data across multiple devices for heavily used databases.

See “Disk I/O management” on page 102 for information about `sp_sysmon` reports on disk utilization.

- Adjusting the fillfactor for create index commands

You may need to adjust the fillfactor in create index commands. Because of the added throughput with multiple processors, setting a lower fillfactor may temporarily reduce contention for the data and index pages.

- Transaction length

Transactions that include many statements or take a long time to run may result in increased lock contention. Keep transactions as short as possible, and avoid holding locks – especially exclusive or update locks – while waiting for user interaction

- Temporary tables

Temporary tables (tables in `tempdb`) do not cause contention, because they are associated with individual users and are not shared. However, if multiple user processes use `tempdb` for temporary objects, there can be some contention on the system tables in `tempdb`.

See “Temporary tables and locking” on page 388 for information on ways to reduce contention.

Distributing Engine Resources

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

Understanding how Adaptive Server uses CPU resources is a prerequisite for understanding this chapter.

For more information, see Chapter 4, “Using Engines and CPUs.”

Topic	Page
Algorithm for successfully distributing engine resources	59
Manage preferred access to resources	67
Types of execution classes	67
Setting execution class attributes	71
Rules for determining precedence and scope	77
Example scenario using precedence rules	82
Considerations for Engine Resource Distribution	85

Algorithm for successfully distributing engine resources

This section gives an approach for successful tuning on the task level.

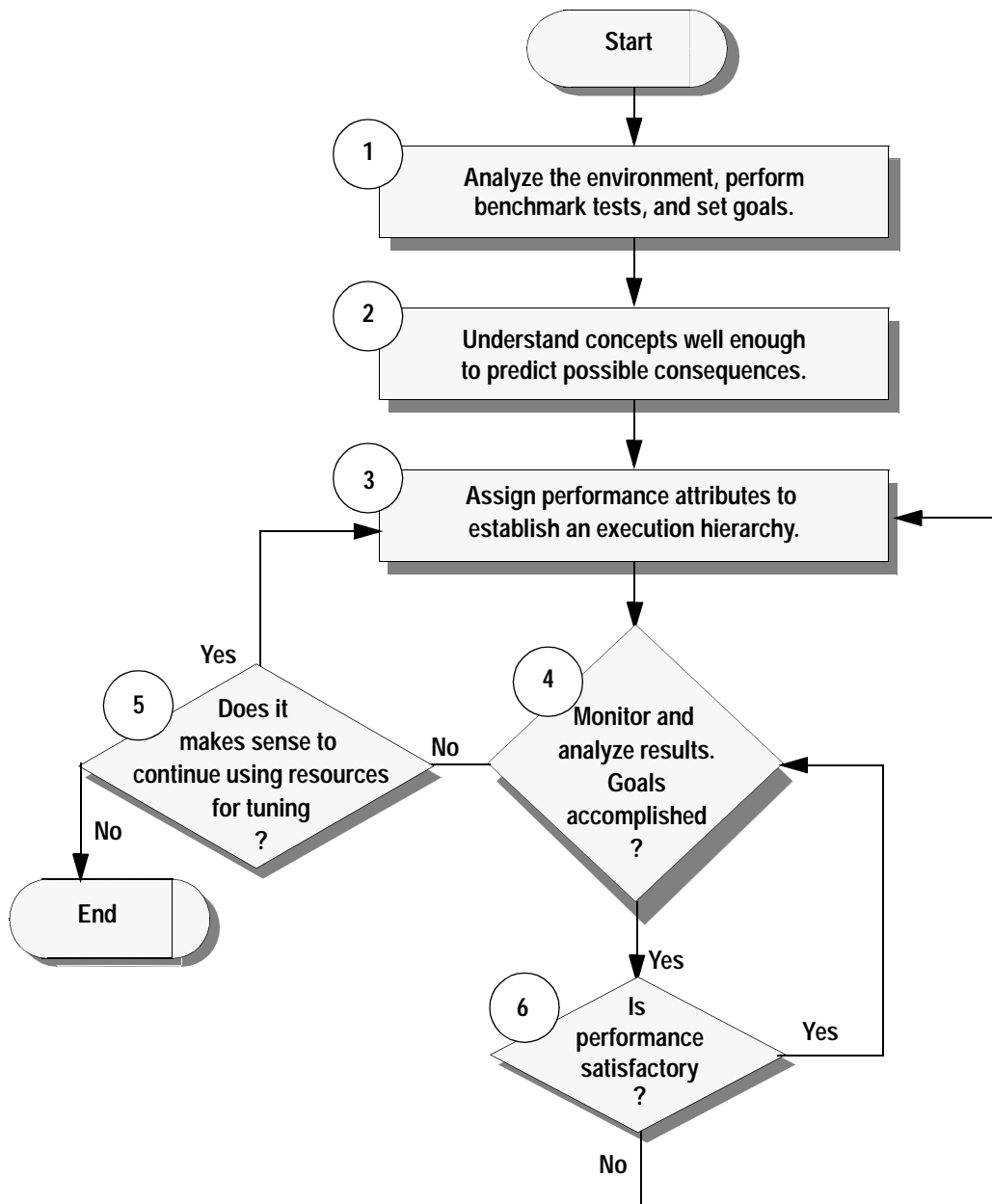
The interactions among execution objects in an Adaptive Server environment are complex. Furthermore, every environment is different: Each involves its own mix of client applications, logins, and stored procedures and is characterized by the interdependencies between these entities.

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

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

Because of the unique nature of every Adaptive Server environment, it is impossible to provide a detailed procedure for assigning execution precedence that makes sense for all systems. However, this section provides guidelines with a progression of steps to use and to discuss the issues commonly related to each step.

The steps involved with assigning execution attributes are illustrated in Figure 5-1. A discussion of the steps follows the figure.

Figure 5-1: Process for assigning execution precedence

Algorithm guidelines

- 1 Study the Adaptive Server environment.
See “Environment analysis and planning” on page 63 for details.
 - Analyze the behavior of all execution objects and categorize them as well as possible.
 - Understand interdependencies and interactions between execution objects.
 - Perform benchmark tests to use as a baseline for comparison after establishing precedence.
 - Think about how to distribute processing in a multiprocessor environment.
 - Identify the critical execution objects for which you will enhance performance.
 - Identify the noncritical execution objects that can afford decreased performance.
 - Establish a set of quantifiable performance goals for the execution objects identified in the last two items.
- 2 Understand the effects of using execution classes.
See “Execution class attributes” on page 69 for details.
 - Understand the basic concepts associated with execution class assignments.
 - Decide whether you need to create one or more user defined-execution classes.
 - Understand the implications of different class level assignments—how do assignments affect the environment in terms of performance gains, losses, and interdependencies?
- 3 Assign execution classes and any independent engine affinity attributes.
- 4 After making execution precedence assignments, analyze the running Adaptive Server environment.
See “Results analysis and tuning” on page 66 for details.
 - Run the benchmark tests you used in step 1 and compare the results.
 - If the results are not what you expect, take a closer look at the interactions between execution objects, as outlined in step 1.

- Investigate dependencies that you might have missed.
- 5 Fine tune the results by repeating steps 3 and 4 as many times as necessary.
- 6 Monitor the environment over time.

Environment analysis and planning

This section elaborates on step 1 of “Algorithm for successfully distributing engine resources” on page 59.

Environment analysis and planning involves the following actions:

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

Analyzing

The degree to which your execution attribute assignments enhance an execution object’s performance is a function of the execution object’s characteristics and its interactions with other objects in the Adaptive Server environment. It is essential to study and understand the Adaptive Server environment in detail so that you can make decisions about how to achieve the performance goals you set.

Where to start

Analysis involves these two phases:

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

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

- It is a highly critical execution object needing enhanced response time,
- It is an execution object of medium importance, or

- It is a noncritical execution object that can afford slower response time.

Example: phase 1 – execution object behavior

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

Intrusive and unintrusive

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

Intrusive applications

Effect of assigning attributes	Assigning high execution attributes to intrusive applications might degrade performance.
--------------------------------	--

Example	Consider a situation in which a noncritical application is ready to release a resource, but becomes blocked when a highly-critical application starts executing. If a second critical application needs to use the blocked resource, then execution of this second critical application is also blocked
---------	---

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

Unintrusive applications

Effect of assigning attributes	You can expect enhanced performance when you assign preferred execution attributes to an unintrusive application.
--------------------------------	---

Example	Simultaneous distinct operations on tables in different databases are unintrusive. Two operations are also unintrusive if one is compute bound and the other is I/O bound.
---------	--

I/O-intensive and CPU-intensive execution objects

When an execution object is I/O intensive, it might help to give it EC1 attributes and, at the same time, assign EC3 attributes to any compute-bound execution objects. This can help because an object performing I/O will not normally use an entire time quantum, and will give up the CPU before waiting for I/O to complete.

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

Example: phase 2 – the environment as a whole

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

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

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

Using Adaptive Server monitoring tools is one of the best ways to understand how execution objects affect the environment.

Performing benchmark tests

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

Two tools that can help you understand system and application behavior are:

- Adaptive Server Monitor provides a comprehensive set of performance statistics. It offers graphical displays through which you can isolate performance problems.
- `sp_sysmon` is a system procedure that monitors system performance for a specified time interval and then prints out an ASCII text-based report.

For information on using `sp_sysmon` see *Performance and Tuning Guide: Monitoring and Analyzing for Performance*. In particular, see “Application management” on page 37.

Setting goals

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

Results analysis and tuning

Here are some suggestions for analyzing the running Adaptive Server environment after you configure the execution hierarchy:

- 1 Run the same benchmark tests you ran before assigning the execution attributes, and compare the results to the baseline results. See “Environment analysis and planning” on page 63.
- 2 Ensure that there is good distribution across all the available engines using Adaptive Server Monitor or `sp_sysmon`. Check the “Kernel Utilization” section of the `sp_sysmon` report.

Also see “Application management” on page 37 in the *Performance and Tuning: Monitoring and Analyzing for Performance*

- 3 If the results are not what you expected, take a closer look at the interactions between execution objects.

As described in “Environment analysis and planning” on page 63, look for inappropriate assumptions and dependencies that you might have missed.

- 4 Make adjustments to the performance attributes.
- 5 Finetune the results by repeating these steps as many times as necessary.

Monitoring the environment over time

Adaptive Server has several stored procedures for example `sp_sysmon`, `optdiag`, `sp_spaceused`, that are used to monitor performance and will give valid information on the status of the system.

See *Performance and Tuning Guide: Tools for Monitoring and Analyzing for Performance* for information on monitoring the system.

Manage preferred access to resources

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

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

You can use system procedures to assign *execution attributes* that indicate which tasks should be given preferred access to resources. The Logical Process Manager uses the execution attributes when it assigns priorities to tasks and tasks to engines.

Execution attributes also affect how long a process can use an engine each time the process runs. In effect, assigning execution attributes lets you suggest to Adaptive Server how to distribute engine resources between client applications, logins, and stored procedures in a mixed workload environment.

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

Warning! Assign execution attributes with caution.

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

Types of execution classes

An *execution class* is a specific combination of execution attributes that specify values for task priority, time slice, and task-to-engine affinity. You can bind an execution class to one or more *execution objects*, which are client applications, logins, and stored procedures.

There are two types of execution classes – *predefined* and *user-defined*. Adaptive Server provides three predefined execution classes. You can create user-defined execution classes by combining execution attributes.

Predefined execution classes

Adaptive Server provides the following predefined execution classes:

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

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

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

To change an execution object's execution class from the EC2 default, use `sp_bindexclass`, described in "Assigning execution classes" on page 72.

User-Defined execution classes

In addition to the predefined execution classes, you can define your own execution classes. Reasons for doing this include:

- EC1, EC2, and EC3 do not accommodate all combinations of attributes that might be useful.
- Associating execution objects with a particular group of engines would improve performance.

The system procedure `sp_addexclass` creates a user-defined execution class with a name and attributes that you choose. For example, the following statement defines a new execution class called DS with a low-priority value and allows it to run on any engine:

```
sp_addexclass DS, LOW, 0, ANYENGINE
```

You associate a user-defined execution class with an execution object using `sp_bindexclass` just as you would with a predefined execution class.

Execution class attributes

Each predefined or user-defined execution class is composed of a combination of three attributes: base priority, time slice, and an engine affinity. These attributes determine performance characteristics during execution.

The attributes for the predefined execution classes, EC1, EC2, and EC3, are fixed, as shown in Table 5-1. You specify the mix of attribute values for user-defined execution classes when you create them, using `sp_addexeclass`.

Table 5-1: Fixed-attribute composition of predefined execution classes

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

See “Base priority” on page 69, “Time slice” on page 70 and “Task-to-engine affinity” on page 70 for more information.

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

Base priority

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

When an engine looks for a task to run, it first checks its own high-priority run queue, then the high-priority global run queue, then its own medium-priority run queue, and so on. The effect is that runnable tasks in the high-priority run queues are scheduled onto engines more quickly, than tasks in the other queues.

During execution, Adaptive Server can temporarily change a task’s priority if it needs to. It can be greater than or equal to, but never lower than, its base priority.

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

Time slice

Adaptive Server handles several processes concurrently by switching between them, allowing one process to run for a fixed period of time (a time slice) before it lets the next process run.

As shown in Table 5-1 on page 69, the time slice attribute is different for each predefined execution class. EC1 has the longest time slice value, EC3 has the shortest time slice value, and EC2 has a time slice value that is between the values for EC1 and EC3.

More precisely, the time period that each task is allowed to run is based on the value for the time slice configuration parameter, as described in “Scheduling client task processing time” on page 42. Using default values for configuration parameters, EC1 execution objects may run for double the time slice value; the time slice of an EC2 execution object is equivalent to the configured value; and an EC3 execution object yields at the first yield point it encounters, often not running for an entire time slice.

If tasks do not yield the engine for other reasons (such as needing to perform I/O or being blocked by a lock) the effect is that EC1 clients run longer and yield the engine fewer times over the life of a given task. EC3 execution objects run for very short periods of time when they have access to the engine, so they yield much more often over the life of the task. EC2 tasks fall between EC1 and EC3 in runtime and yields.

Currently, you cannot assign time slice values when you create user-defined execution classes with `sp_addexclass`. Adaptive Server assigns the EC1, EC2, and EC3 time slice values for high, medium, and low priority tasks, respectively.

Task-to-engine affinity

In a multiengine environment, any available engine can process the next task in the global run queue. The engine affinity attribute lets you assign a task to an engine or to a group of engines. There are two ways to use task-to-engine affinity:

- Associate less critical execution objects with a defined group of engines to restrict the object to a subset of the total number of engines. This reduces processor availability for those objects. The more critical execution objects can execute on any Adaptive Server engine, so performance for them improves because they have the benefit of the resources that the less critical ones are deprived of.

- Associate more critical execution objects with a defined group of engines to which less critical objects do not have access. This ensures that the critical execution objects have access to a known amount of processing power.

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

You can create engine groups with `sp_addengine` and bind execution objects to an engine group with `sp_addexclass`. If you do not want to assign engine affinity for a user-defined execution class, using `ANYENGINE` as the engine group parameter allows the task to run on any engine.

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

Setting execution class attributes

You implement and manage execution hierarchy for client applications, logins, and stored procedures using the five categories of system procedures listed in the following table.

Table 5-2: System procedures for managing execution object precedence

Category	Description	System procedures
User-defined execution class	Create and drop a user-defined class with custom attributes or change the attributes of an existing class.	<ul style="list-style-type: none"> • <code>sp_addexclass</code> • <code>sp_dropexclass</code>
Execution class binding	Bind and unbind predefined or user-defined classes to client applications and logins.	<ul style="list-style-type: none"> • <code>sp_bindexclass</code> • <code>sp_unbindexclass</code>
For the session only (“on the fly”)	Set and clear attributes of an active session only.	<ul style="list-style-type: none"> • <code>sp_setpsexe</code> • <code>sp_clearpsexe</code>
Engines	Add engines to and drop engines from engine groups; create and drop engine groups.	<ul style="list-style-type: none"> • <code>sp_addengine</code> • <code>sp_dropengine</code>
Reporting	Report on engine group assignments, application bindings, execution class attributes.	<ul style="list-style-type: none"> • <code>sp_showcontrolinfo</code> • <code>sp_showexclass</code> • <code>sp_showpsexe</code>

See the *Adaptive Server Enterprise Reference Manual* for complete descriptions of the system procedures in Table 5-2.

Assigning execution classes

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

The syntax for the `sp_bindexeclass` is:

```
sp_bindexeclass object_name, object_type,  
                scope, class_name
```

Suppose you decide that the “sa” login must get results from isql as fast as possible. You can tell Adaptive Server to give execution preference to login “sa” when it executes isql by issuing `sp_bindexeclass` with the preferred execution class EC1. For example:

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

This statement stipulates that whenever a login (LG) called “sa” executes the isql application, the “sa” login task executes with EC1 attributes. Adaptive Server improves response time for the “sa” login by:

- Placing it in a high-priority run queue, so it is assigned to an engine more quickly
- Allowing it to run for a longer period of time than the default value for time slice, so it accomplishes more work when it has access to the engine

Engine groups and establishing task-to-engine affinity

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

The example uses `sp_addengine` and `sp_addexeclass`.

You create engine groups and add engines to existing groups with `sp_addengine`. The syntax is:

```
sp_addengine engine_number, engine_group
```

You set the attributes for user-defined execution classes using `sp_addexclass`. The syntax is:

```
sp_addexclass class_name, base_priority,
              time_slice, engine_group
```

The steps are:

- 1 Create an engine group using `sp_addengine`. This statement creates a group called `DS_GROUP`, consisting of engine 3:

```
sp_addengine 3, DS_GROUP
```

To expand the group so that it also includes engines 4 and 5, execute `sp_addengine` two more times for those engine numbers:

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

- 2 Create a user-defined execution class and associate it with the `DS_GROUP` engine group using `sp_addexclass`.

This statement defines a new execution class called `DS` with a priority value of “LOW” and associates it with the engine group `DS_GROUP`:

```
sp_addexclass DS, LOW, 0, DS_GROUP
```

- 3 Bind the less critical execution objects to the new execution class using `sp_bindexclass`.

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

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

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

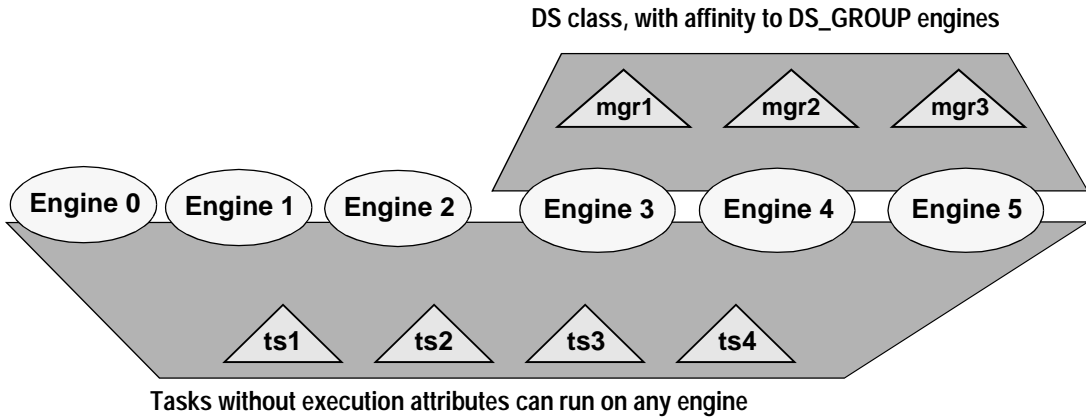
The result of this example is that the technical support group (not bound to an engine group) is given access to more immediate processing resources than the managers.

Figure 5-2 illustrates the associations in this scenario:

- Logins “mgr1”, “mgr2”, and “mgr3” have affinity to the `DS` engine group consisting of engines 3, 4, and 5.

- Logins “ts1”, “ts2”, “ts3”, and “ts4” can use all six Adaptive Server engines.

Figure 5-2: An example of engine affinity



How execution class bindings affect scheduling

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

- An `order_entry` application, an OLTP application critical to taking customer orders.
- A `sales_report` application, that can prepare various reports. Some managers run this application with default characteristics, but other managers run the report at lower priority.
- Other users, who are running various other applications at default priorities (no assignment of execution classes or priorities).

Execution class bindings

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

```
sp_bindexeclclass order_entry, AP, NULL, EC1
```

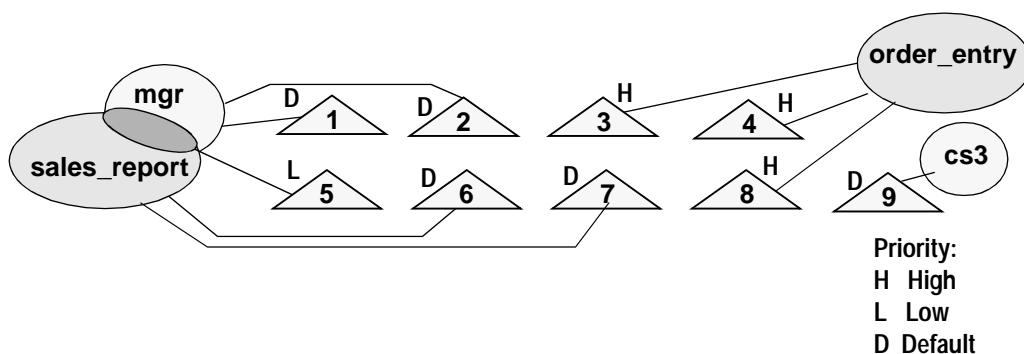

The following `sp_bindexeclass` statement specifies EC3 when “mgr” runs the `sales_report` application:

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

This task can execute only when tasks with EC1 and EC2 attributes are idle or in a sleep state.

Figure 5-3 shows four execution objects running tasks. Several users are running the `order_entry` and `sales_report` applications. Two other logins are active, “mgr” (logged in once using the `sales_report` application, and twice using `isql`) and “cs3” (not using the affected applications).

Figure 5-3: Execution objects and their tasks



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

Engine affinity can affect scheduling

Each execution class is associated with a different priority:

- Tasks assigned to EC1 are placed in a high-priority run queue.
- Tasks assigned to EC2 are placed in a medium-priority run queue.
- Tasks assigned to EC3 are placed in a low-priority run queue.

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

What happens if a task has affinity to a particular engine? Assume that task 7 in Figure 5-3 on page 75, a high-priority task in the global run queue, has a user-defined execution class with high priority and affinity to engine 2. Engine 2 currently has high-priority tasks queued and is running another task.

If engine 1 has no high-priority tasks queued when it finishes processing task 8 in Figure 5-3 on page 75, it checks the global run queue, but cannot process task 7 due to the engine binding. Engine 1 then checks its own medium-priority queue, and runs task 15. Although a System Administrator assigned the preferred execution class EC1, engine affinity temporarily lowered task 7's execution precedence to below that of a task with EC2.

This effect might be highly undesirable or it might be what the performance tuner intended. You can assign engine affinity and execution classes in such a way that task priority is not what you intended. You can also make assignments in such a way that tasks with low priority might not ever run, or might wait for extremely long times – another reason to plan and test thoroughly when assigning execution classes and engine affinity.

Setting attributes for a session only

If you need to change any attribute value temporarily for an active session, you can do so using `sp_setpsex`.

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

To clear attributes set for a session, use `sp_clearpsex`.

Getting information

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

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

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

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

Rules for determining precedence and scope

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

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

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

Multiple execution objects and ECs

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

Use the rules in this order:

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

Precedence rule

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

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

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

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

Precedence Rule Example

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

The login's attributes override those of the client application, so the login is given preference for processing. If the stored procedure has a higher base priority than the login, the base priority of the Adaptive Server process executing the stored procedure goes up temporarily for the duration of the stored procedure's execution. Figure 5-4 shows how the precedence rule is applied.

Figure 5-4: Use of the precedence rule



Stored procedure runs with EC2

What happens when a login with EC2 invokes a client application with EC1 and the client application calls a stored procedure with EC3? The stored procedure executes with the attributes of EC2 because the execution class of a login precedes that of a client application.

Scope rule

In addition to specifying the execution attributes for an object, you can define its scope when you use `sp_bindexeclass`. The scope specifies the entities for which the execution class bindings will be effective. The syntax is:

```
sp_bindexeclass object_name, object_type,  
                scope, class_name
```

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

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

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

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

The execution object’s execution attributes apply to all of its interactions if the scope is NULL.

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

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

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

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

Combined with the bindings above that grant the “sa” user of isql EC1 execution attributes, and using the precedence rule, an isql request from the “sa” login executes with EC1 attributes. Other processes servicing isql requests from non-“sa” logins execute with EC3 attributes.

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

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

Resolving a precedence conflict

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

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

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

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

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

Examples: determining precedence

Each row in Table 5-3 contains a combination of execution objects and their conflicting execution attributes.

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

The remaining columns show how Adaptive Server resolves precedence.

Table 5-3: Conflicting attribute values and Adaptive Server assigned values

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

To test your understanding of the rules of precedence and scope, cover the “Adaptive Server-Assigned Values” columns in Table 5-3, and predict the values in those columns. Following is a description of the scenario in the first row, to help get you started:

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

At run time:

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

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

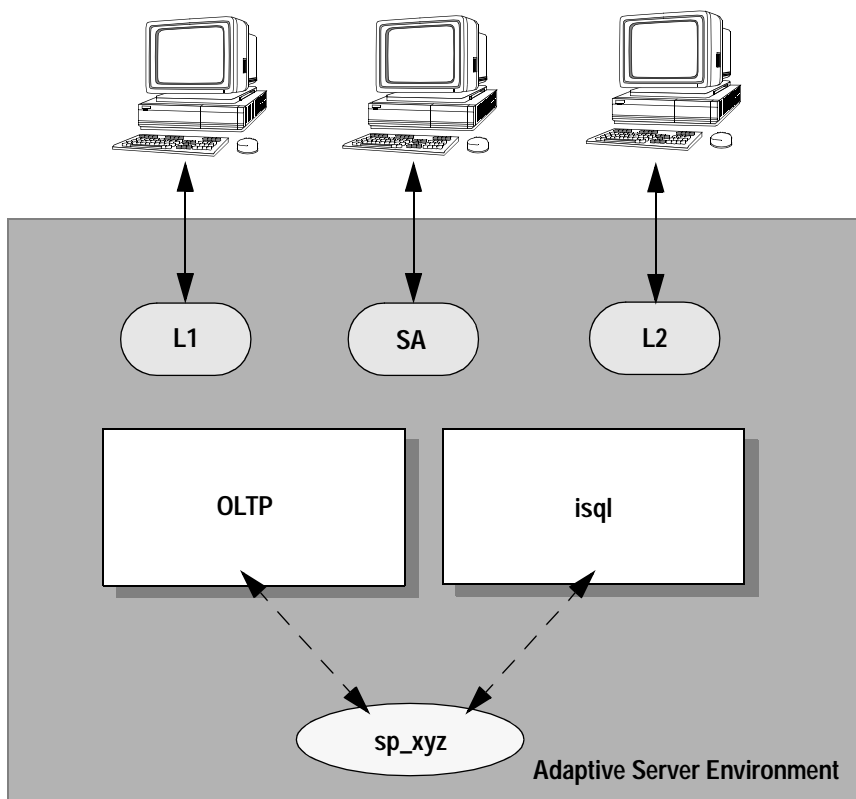
Example scenario using precedence rules

This section presents an example that illustrates how Adaptive Server interprets the execution class attributes.

Figure 5-5 shows two client applications, OLTP and isql, and three Adaptive Server logins, “L1”, “sa”, and “L2”.

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

Figure 5-5: Conflict resolution



The rest of this section describes one way to implement the steps discussed in Algorithm Guidelines.

Planning

The System Administrator performs the analysis described in steps 1 and 2 of the algorithm in “Algorithm for successfully distributing engine resources” on page 59 and decides on the following hierarchy plan:

- The OLTP application is an EC1 application and the isql application is an EC3 application.

- Login “L1” can run different client applications at different times and has no special performance requirements.
- Login “L2” is a less critical user and should always run with low performance characteristics.
- Login “sa” must always run as a critical user.
- Stored procedure sp_xyz should always run with high performance characteristics. Because the isql client application can execute the stored procedure, giving sp_xyz a high-performance characteristics is an attempt to avoid a bottleneck in the path of the OLTP client application.

Table 5-1 summarizes the analysis and specifies the execution class to be assigned by the System Administrator. Notice that the tuning granularity gets finer as you descend the table. Applications have the greatest granularity, or the largest scope. The stored procedure has the finest granularity, or the narrowest scope.

Table 5-4: Example analysis of an Adaptive Server environment

Identifier	Interactions and comments	Execution class
OLTP	<ul style="list-style-type: none"> • Same tables as isql • Highly critical 	EC1
isql	<ul style="list-style-type: none"> • Same tables as OLTP • Low priority 	EC3
L1	<ul style="list-style-type: none"> • No priority assignment 	None
sa	<ul style="list-style-type: none"> • Highly critical 	EC1
L2	<ul style="list-style-type: none"> • Not critical 	EC3
sp_xyz	<ul style="list-style-type: none"> • Avoid “hot spots” 	EC1

Configuration

The System Administrator executes the following system procedures to assign execution classes (algorithm step 3):

```

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

Execution characteristics

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

- 1 A client logs in to Adaptive Server as “L1” using OLTP.
 - Adaptive Server determines that OLTP is EC1.
 - “L1” does not have an execution class, so Adaptive Server assigns the default class EC2. “L1” gets the characteristics defined by EC1 when it invokes OLTP.
 - If “L1” executes stored procedure sp_xyz, its priority remains unchanged while sp_xyz executes. During execution, “L1” has EC1 attributes throughout.
- 2 A client logs in to Adaptive Server as “L1” using isql.
 - Because isql is EC3, and the “L1” execution class is undefined, “L1” executes with EC3 characteristics. This means it runs at low priority and has affinity with the highest numbered engine (as long as there are multiple engines).
 - When “L1” executes sp_xyz, its priority is raised to high because the stored procedure is EC1.
- 3 A client logs in to Adaptive Server as “sa” using isql.
 - Adaptive Server determines the execution classes for both isql and the “sa”, using the precedence rule. Adaptive Server runs the System Administrator’s instance of isql with EC1 attributes. When the System Administrator executes sp_xyz, the priority does not change.
- 4 A client logs in to Adaptive Server as “L2” using isql.
 - Because both the application and login are EC3, there is no conflict. “L2” executes sp_xyz at high priority.

Considerations for Engine Resource Distribution

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

Table 5-5: When assigning execution precedence is useful

Execution object	Description
Client application	There is little contention for non-CPU resources among client applications.
Adaptive Server login	One login should have priority over other logins for CPU resources.
Stored procedure	There are well-defined stored procedure “hot spots.”

It is more effective to lower the execution class of less-critical execution objects than to raise the execution class of a highly critical execution object. The sections that follow give more specific consideration to improving performance for the different types of execution objects.

Client applications: OLTP and DSS

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

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

Unintrusive client applications

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

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

I/O-bound client applications

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

An I/O-bound process, on the other hand, gives up the CPU each time it performs an I/O operation. Assigning a non-preferred execution class to the compute-bound application enables Adaptive Server to run the I/O-bound process sooner.

Highly critical applications

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

Adaptive Server logins: high-priority users

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

Stored procedures: “hot spots”

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

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

Controlling Physical Data Placement

This describes how controlling the location of tables and indexes can improve performance.

Topic	Page
Object placement can improve performance	89
Terminology and concepts	92
Guidelines for improving I/O performance	92
Using serial mode	96
Creating objects on segments	96
Partitioning tables for performance	99
Space planning for partitioned tables	103
Commands for partitioning tables	106
Steps for partitioning tables	117
Special procedures for difficult situations	124
Maintenance issues and partitioned tables	131

Object placement can improve performance

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

- Place a database's data segments on a specific device or devices, storing the database's log on a separate physical device. This way, reads and writes to the database's 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 itself. 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.

For multiuser systems and multi-CPU systems that perform a lot of disk I/O, pay special attention to physical and logical device issues and the distribution of I/O across devices:

- Plan 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.
- Use a number of partitions that will allow parallel scans, to meet query performance goals.
- Make use of the ability to create database to perform parallel I/O on as many as six devices at a time, to gain a significant performance leap for creating multi gigabyte databases.

Symptoms of poor object placement

The following symptoms may indicate that your system could benefit from attention to object placement:

- Single-user performance is satisfactory, but response time increases significantly when multiple processes are executed.
- Access to a mirrored disk takes twice as long as access to an unmirrored disk.
- Query performance degrades as system table activity increases.
- Maintenance activities seem to take a long time.
- Stored procedures seem to slow down as they create temporary tables.
- 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.

Underlying problems

If you are experiencing problems due to disk contention and other problems related to object placement, check for these underlying problems:

- Random-access (I/O for data and indexes) and serial-access (log I/O) processes are using the same disks.
- Database processes and operating system processes are using the same disks.
- Serial disk mirroring is being used because of functional requirements.
- Database maintenance activity (logging or auditing) is taking place on the same disks as data storage.
- tempdb activity is on the same disk as heavily used tables.

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.

For more information about using `sp_sysmon`, see Chapter 8, “Monitoring Performance with `sp_sysmon`.” in the *Performance and Tuning: Monitoring and Analyzing for Performance* book.

Pay special attention to the output associated with the discussions:

- I/O device contentions
- APL heap tables
- Last page locks on heaps
- Disk I.O management

Adaptive Server Monitor can also help pinpoint problems.

Terminology and concepts

You should understand the following distinctions between logical or database devices and physical devices:

- The *physical disk* or *physical device* is the actual hardware that stores the data.
- A *database device* or *logical device* is a piece of 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 Adaptive Server installation and configuration guides 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 block of storage for a table. Partitioning a table splits it so that multiple tasks can access it simultaneously. When partitioned tables are placed on segments with a matching number of devices, each partition starts on a separate database device.

Use `sp_helpdevice` to get information about devices, `sp_helpsegment` to get information about segments, and `sp_helppartition` to get information about partitions.

Guidelines for improving I/O performance

The major guidelines for improving I/O performance in Adaptive Server are as follows:

- 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.
- Partitioning tables to match the number of physical devices in a segment.

Spreading data across disks to avoid I/O contention

You can avoid bottlenecks by spreading data storage across multiple disks and multiple disk controllers:

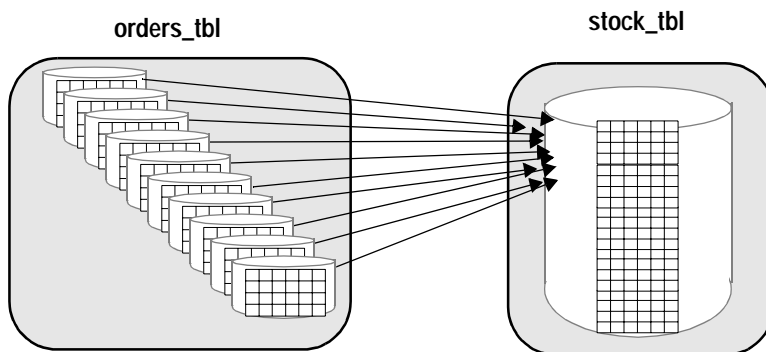
- Put databases with critical performance requirements on separate devices. If possible, also use separate controllers from those used by other databases. Use segments as needed for critical tables and partitions as needed for parallel queries.
- Put heavily used tables on separate disks.
- Put frequently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

Avoiding physical contention in parallel join queries

The example in Figure 6-1 illustrates a join of two partitioned tables, `orders_tbl` and `stock_tbl`. There are ten worker processes available: `orders_tbl` has ten partitions on ten different physical devices and is the outer table in the join; `stock_tbl` is nonpartitioned. The worker processes will not have a problem with access contention on `orders_tbl`, but each worker process must scan `stock_tbl`. There could be a problem with physical I/O contention if the entire table does not fit into a cache. In the worst case, ten worker processes attempt to access the physical device on which `stock_tbl` resides. You can 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 6-1: Joining tables on different physical devices



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

Place system databases with heavy I/O requirements on separate physical disks and controllers than your application databases.

Where to place *tempdb*

tempdb is automatically installed on the master device. If more space is needed, *tempdb* can be expanded to other devices. If *tempdb* is expected to be quite active, place it on a disk that is not used for other important database activity. Use the fastest disk available for *tempdb*. It is a heavily used database that affects all processes on the server.

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

See Chapter 17, “*tempdb* Performance Issues,” for more placement issues and performance tips for *tempdb*.

Where to place *sybsecurity*

If you use auditing on your Adaptive Server, 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 not used for tables where fast response time is critical. Placing *sybsecurity* on its own device is optimal.

Also, use the threshold manager to monitor its free space to avoid suspending user transactions if the audit database fills up.

Keeping transaction logs on a separate disk

You can limit the size of the transaction logs by placing them on a separate segment, this keeps it from competing with other objects for disk space. Placing the log on a separate physical disk:

- Improves performance by reducing I/O contention
- Ensures full recovery in the event of hard disk crashes 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

Placing the transaction log on the same device as the data itself causes such a dangerous reliability problem that both `create database` and `alter database` require the use of the `with override` option to put the transaction log on the same device as the data itself.

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 for deferred updates or transaction rollbacks.

If your 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, the extents tend to be allocated sequentially, reducing disk head travel and seeks, thereby maintaining a higher I/O rate.

Also, if log and data are on separate devices, Adaptive Server buffers log records for each user in a user log cache, reducing 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 penalty on SMP systems.

If you have created a database without its log on a separate device, see the *System Administration Guide*.

Mirroring a device on a separate disk

If you mirror data, put the mirror on a separate physical disk than the device that it mirrors. Disk hardware failure often results in whole physical disks being lost or unavailable. Mirroring on separate disks also minimizes the performance impact of mirroring.

Device mirroring performance issues

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

See the *System Administration Guide* for more information on mirroring.

If you do not use mirroring, or use operating system mirroring, set the configuration parameter `disable disk mirroring` to 1. This may yield slight performance improvements.

Mirroring can slow the time taken to complete disk writes, since writes go to both disks, either serially or simultaneously. Reads always come from the primary side. 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 are started at the same time, and Adaptive Server waits for both to complete. The time to complete nonserial writes is $\max(W1, W2)$ – 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 $W1 + W2$ – the sum of the two I/O times.

Using serial mode

Despite its performance impact, serial mode is important for reliability. In fact, serial mode is the default, 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, it is impossible for a single failure to affect both disks. Specifying nonserial mode improves performance, but you risk losing data if a failure occurs that affects both writes.

Warning! Unless you are sure that your mirrored database system does not need to be absolutely reliable, do not use nonserial mode.

Creating objects on segments

A segment is a label that points to one or more database devices.

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. Segments label space on one or more logical devices.

Tables and indexes are stored on segments. If no segment is named in the create table or create index statement, then the objects are stored on the default segment for the database. Naming a segment in either of these commands creates the object on the segment. The `sp_placeobject` system procedure causes 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 the disk must be allocated to the database by the System Administrator or the Database Owner with `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.

If you create a user-defined segment, you can place tables or indexes on that segment with the `create table` or `create index` commands:

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

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

Using segments

Segments can 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
- Placing the text and image page chain on a separate disk from the table itself, where the pointers to the text values are stored

In addition, segments can control space usage, as follows:

- A table can never grow larger than its segment allocation; You can use segments to limit table size.
- Tables on other segments cannot impinge on the space allocated to objects on another segment.
- The threshold manager can monitor space usage.

Separating tables and indexes

Use segments to isolate tables on one set of disks and nonclustered indexes on another set of disks. You cannot place a clustered index on a separate 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 there.

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

Splitting large tables across devices

Segments can span multiple devices, so they can be used to spread data across one or more disks. For large, extremely busy tables, this can help balance the I/O load. For parallel queries, creating segments that include multiple devices is essential for I/O parallelism during partitioned-based scans.

See the *System Administration Guide* for more information.

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 LOB (large object) chain.

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 these 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` 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. Note that 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.

You can use `sp_placeobject` to move all future allocations for the LOB columns to a separate segment.

See the *System Administration Guide* for more information.

Partitioning tables for performance

Partitioning a table can improve performance for several types of processes. The reasons for partitioning a table are:

- 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 makes it possible to load a table in parallel with bulk copy. For more information on parallel bcp, see the *Utility Programs* manual.
- Partitioning makes it possible to distribute a table's I/O over multiple database devices.
- Partitioning provides multiple insertion points for a heap table.

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

The following sections explain the commands needed to partition tables and to maintain partitioned tables, and outline the steps for different situations.

See “Guidelines for parallel query configuration” on page 164 in the *Performance and Tuning: Optimizer* book for more information and examples of partitioning to meet specific performance goals.

User transparency

Adaptive Server's management of partitioned tables is transparent to users and applications. Partitioned tables do not appear different from nonpartitioned tables when queried or viewed with most utilities. Exceptions are:

- If queries do not include order by or other commands that require a sort, data returned by a parallel query may not in the same order as data returned by serial queries.
- The dbcc checktable and dbcc checkdb commands list the number of data pages in each partition.

See the *System Administration Guide* for information about dbcc.

- `sp_helpartition` lists information about a table's partitions.
- `showplan` output displays messages indicating the number of worker processes used for queries that are executed in parallel, and the statistics io "Scan count" shows the number of scans performed by worker processes.
- Parallel bulk copy allows you to copy to a particular partition of a heap table.

Partitioned tables and parallel query processing

Parallel query processing on partitioned tables can potentially produce dramatic improvements in query performance. Partitions increase simultaneous access by worker processes. When enough worker processes are available, and the value for the `max parallel degree` configuration parameter is set equal to or greater than the number of partitions, one worker process scans each of the table's partitions.

When the partitions are distributed across physical disks, the reduced I/O contention further speeds parallel query processing and achieves a high level of parallelism.

The optimizer can choose to use parallel query processing for a query against a partitioned table when parallel query processing is enabled. The optimizer considers a parallel partition scan for a query when the base table for the query is partitioned, and it considers a parallel index scan for a useful index.

See Chapter 8, "Parallel Query Optimization," in the *Performance and Tuning: Optimizer* book for more information on how parallel queries are optimized.

Distributing data across partitions

Creating a clustered index on a partitioned table redistributes the table's data evenly over the partitions. Adaptive Server determines the index key ranges for each partition so that it can distribute the rows equally in the partition. Each partition is assigned at least one exclusive device if the number of devices in the segment is equal to or greater than the number of partitions.

If you create the clustered index on an empty partitioned table, Adaptive Server prints a warning advising you to re-create the clustered index after loading data into the table, as all the data will be inserted into the first partition until you re-create the clustered index.

If you partition a table that already has a clustered index, all pages in the table are assigned to the first partition. The `alter table...partition` command succeeds and prints a warning. You must drop and recreate the index to redistribute the data.

Improving insert performance with partitions

All insert commands on an allpages-locked heap table attempt to insert the rows on the last page of the table. If multiple users insert data simultaneously, each new insert transaction must wait for the previous transaction to complete in order to proceed.

Partitioning an allpages-locked heap table improves the performance of concurrent inserts by reducing contention for the last page of a page chain.

For data-only-locked tables, Adaptive Server stores one or more hints that point to a page where an insert was recently performed. Blocking during inserts on data-only-locked tables occurs only with high rates of inserts.

Partitioning data-only-locked heap tables increases the number of hints, and can help if inserts are blocking.

How partitions address page contention

When a transaction inserts data into a partitioned heap table, Adaptive Server randomly assigns the transaction to one of the table's partitions. Concurrent inserts are less likely to block, since multiple last pages are available for inserts.

Selecting heap tables to partition

Allpages-locked heap tables that have large amounts of concurrent insert activity will benefit from partitioning. Insert rates must be very high before significant blocking takes place on data-only-locked tables. If you are not sure whether the tables in your database system might benefit from partitioning:

- Use `sp_sysmon` to look for last page locks on heap tables.

See “Lock management” on page 73 in the *Performance and Tuning: Monitoring and Analyzing for Performance* book.

- Use `sp_object_stats` to report on lock contention.

See “Identifying tables where concurrency is a problem” on page 88 in the *Performance and Tuning: Locking* book.

Restrictions on partitioned tables

You cannot partition Adaptive Server system tables or tables that are already partitioned. Once you have partitioned a table, you cannot use any of the following Transact-SQL commands on the table until you unpartition it:

- `sp_placeobject`
- `truncate table`
- `alter table table_name partition n`

See “alter table...unpartition Syntax” on page 107 for more information.

Partition-related configuration parameters

If you require a large number of partitions, you may want to change the default values for the partition groups and partition spinlock ratio configuration parameters.

See the *System Administration Guide* for more information.

How Adaptive Server distributes partitions on devices

When you issue an `alter table...partition` command, Adaptive Server creates the specified number of partitions in the table and distributes those partitions over the database devices in the table’s segment. Adaptive Server assigns partitions to devices so that they are distributed evenly across the devices in the segment.

Table 6-1 illustrates how Adaptive Server assigns 5 partitions to 3, 5, and 12 devices, respectively.

Table 6-1: Assigning partitions to segments

Partition ID	Device (D) Assignments for Segment With		
	3 Devices	5 Devices	12 Devices
Partition 1	D1	D1	D1, D6, D11
Partition 2	D2	D2	D2, D7, D12
Partition 3	D3	D3	D3, D8, D11
Partition 4	D1	D4	D4, D9, D12
Partition 5	D2	D5	D5, D10, D11

Matching the number of partitions to the number of devices in the segment provides the best I/O performance for parallel queries.

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

Table 6-1 and other statements in this chapter describe the Adaptive Server logical devices that map to a single physical device.

A striped RAID device may contain multiple physical disks, but it appears to Adaptive Server as a single logical device. For a striped RAID 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 NT) 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 planning for partitioned tables, the two major issues are:

- Maintaining load balance across the disk for partition-based scan performance and for I/O parallelism
- Maintaining clustered indexes requires approximately 120% of the space occupied by the table to drop and re-create the index or to run reorg rebuild

How you make these decisions depends on:

- The availability of disk resources for storing tables
- The nature of your application mix

You need to estimate how often your partitioned tables need maintenance: some applications need frequent index re-creation to maintain balance, while others need little maintenance.

For those applications that need frequent load balancing for performance, having space to re-create a clustered index or run reorg rebuild provides the speediest and easiest method. 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 356 for more information.

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 “Steps for partitioning tables” on page 117 for more 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 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 size varies, depending on the length of the key. Loading the data into the table initially 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 will be 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. Extent stealing can be detected by using `sp_helpsegment` to check for devices that have no space available and by using `sp_helppartition` 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 “Steps for partitioning tables” on page 117.

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

Take care with large bulk copy in operations. You can use parallel bulk copy to send rows to the partition with the smallest number of pages to balance the data across the partitions. See “Using `bcp` to correct partition balance” on page 112.

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.

Once the fragmentation starts to take its toll on application performance, you need to perform maintenance. If that requires dropping and re-creating the clustered index, you need 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.

Commands for partitioning tables

Creating and maintaining partitioned tables involves using a mix of the following types of commands:

- Commands to partition and unpartition the table
- Commands to drop and re-create clustered indexes to maintain data distribution on the partitions and/or on the underlying physical devices
- Parallel bulk copy commands to load data into specific partitions
- Commands to display information about data distribution on partitions and devices
- Commands to update partition statistics

This section presents the syntax and examples for the commands you use to create and maintain partitioned tables.

For different scenarios that require different combinations of these commands, see “Steps for partitioning tables” on page 117.

Use the `alter table` command to partition and unpartition a table.

alter table...partition syntax

The syntax for using the `partition` clause to alter table is:

```
alter table table_name partition n
```

where *table_name* is the name of the table and *n* is the number of partitions you are creating.

Any data that is in the table before you invoke `alter table` remains in the first partition. Partitioning a table does not move the table's data – it will still occupy the same space on the physical devices.

If you are creating partitioned tables for parallel queries, you may need to redistribute the data, either by creating a clustered index or by copying the data out, truncating the table, and then copying the data back in.

You cannot include the `alter table...partition` command in a user-defined transaction.

The following command creates 10 partitions for a table named `historytab`:

```
alter table historytab partition 10
```

***alter table...unpartition* Syntax**

Unpartitioning a table concatenates the table's multiple partitions into a single partition. Unpartitioning a table does not change the location of the data.

The syntax for using the `unpartition` clause to `alter table` is:

```
alter table table_name unpartition
```

For example, to unpartition a table named `historytab`, enter:

```
alter table historytab unpartition
```

Changing the number of partitions

To change the number of partitions in a table, first unpartition the table using `alter table...unpartition`.

Then use `alter table...partition`, specifying the new number of partitions. This does not move the existing data in the table.

You cannot use the `partition` clause with a table that is already partitioned.

For example, if a table named `historytab` contains 10 partitions, and you want the table to have 20 partitions, enter these commands:

```
alter table historytab unpartition
alter table historytab partition 20
```

Distributing data evenly across partitions

Good parallel performance depends on a fairly even distribution of data on a table's partitions. The two major methods to achieve this distribution are:

- Creating a clustered index on a partitioned table. The data should already be in the table.
- Using parallel bulk copy, specifying the partitions where the data is to be loaded.

`sp_helpartition tablename` reports the number of pages on each partition in a table.

Commands to create and drop clustered indexes

You can create a clustered index using the `create clustered index` command or by creating a primary or foreign key constraint with `alter table...add constraint`. The steps to drop and re-create it are slightly different, depending on which method you used to create the existing clustered index.

Creating a clustered index on a partitioned table requires a parallel sort. Set configuration parameters and set options as shown before you issue the command to create the index:

- Set number of worker processes and max parallel degree to at least the number of partitions in the table, plus 1.
- Execute `sp_dboption "select into/bulkcopy/pllsort", true`, and run checkpoint in the database.

For more information on configuring Adaptive Server to allow parallel execution, see “Controlling the degree of parallelism” on page 154 in the *Performance and Tuning: Optimizer* book.

See Chapter 9, “Parallel Sorting,” in the *Performance and Tuning: Optimizer* book for additional information on parallel sorting.

If your queries do not use the clustered index, you can drop the index without affecting the distribution of data. Even if you do not plan to retain the clustered index, be sure to create it on a key that has a very high number of data values. For example, a column such as “sex”, which has only the values “M” and “F”, will not provide a good distribution of pages across partitions.

Creating an index using parallel sort is a minimally logged operation and is not recoverable. You should dump the database when the command completes.

Using *reorg rebuild* on data-only-locked tables

The *reorg rebuild* command copies data rows in data-only-locked tables to new data pages. If there is a clustered index, rows are copied in clustered key order.

Running *reorg rebuild* redistributes data evenly on partitions. The clustered index and any nonclustered indexes are rebuilt. To run *reorg rebuild* on the table, provide only the table name:

```
reorg rebuild titles
```

Using *drop index* and *create clustered index*

If the index on the table was created with *create index*:

- 1 Drop the index:

```
drop index huge_tab.cix
```

- 2 Create the clustered index, specifying the segment:

```
create clustered index cix
on huge_tab(key_col)
on big_demo_seg
```

Using constraints and *alter table*

If the index on the table was created using a constraint, follow these steps to re-create a clustered index:

- 1 Drop the constraint:

```
alter table huge_tab drop constraint prim_key
```

- 2 Re-create the constraint, thereby re-creating the index:

```
alter table huge_tab add constraint prim_key
primary key clustered (key_col)
on big_demo_seg
```

Special concerns for partitioned tables and clustered indexes

Creating a clustered index on a partitioned table is the only way to redistribute data on partitions without reloading the data by copying it out and back into the table.

When you are working with partitioned tables and clustered indexes, there are two special concerns:

- Remember that the data in a clustered index “follows” the index, and that if you do not specify a segment in create index or alter table, the default segment is used as the target segment.
- You can use the with sorted_data clause to avoid sorting and copying data while you are creating a clustered index. This saves time when the data is already in clustered key order. However, when you need to create a clustered index to load balance the data on partitions, do not use the sorted_data clause.

See “Creating an index on sorted data” on page 345 for options.

Using parallel *bcp* to copy data into partitions

Loading data into a partitioned table using parallel bcp lets you direct the data to a particular partition in the table.

- Before you run parallel bulk copy, the table should be located on the segment, and it should be partitioned.
- You should drop all indexes, so that you do not experience failures due to index deadlocks.
- Use alter table...disable trigger so that fast, minimally-logged bulk copy is used, instead of slow bulk copy, which is completely logged.
- You may also want to set the database option trunc log on chkpt to keep the log from filling up during large loads.
- You can use operating system commands to split the file into separate files, and then copy each file, or use the -F (first row) and -L (last row) command-line flags for bcp.

Whichever method you choose, be sure that the number of rows sent to each partition is approximately the same.

Here is an example using separate files:

```
bcp mydb..huge_tab:1 in bigfile1
bcp mydb..huge_tab:2 in bigfile2
...
bcp mydb..huge_tab:10 in bigfile10
```

This example uses the first row and last row command-line arguments on a single file:

```
bcp mydb..huge_tab:1 in bigfile -F1 -L100000
bcp mydb..huge_tab:2 in bigfile -F100001 -L200000
```

```
...
bcp mydb..huge_tab:10 in bigfile -F900001 -L1000000
```

If you have space to split the file into multiple files, copying from separate files is much faster than using the first row and last row command-line arguments, since `bcp` needs to parse each line of the input file when using `-F` and `-L`. This parsing process can be very slow, almost negating the benefits from parallel copying.

Parallel copy and locks

Starting many current parallel `bcp` sessions may cause Adaptive Server to run out of locks.

When you copy in to a table, `bcp` acquires an exclusive intent lock on the table, and either page or row locks, depending on the locking scheme. If you are copying in very large tables, and especially if you are performing simultaneous copies into a partitioned table, this can require a very large number of locks.

To avoid running out of locks:

- Set the number of locks configuration parameter high enough, or
- Use the `-b batchsize` `bcp` flag to copy smaller batches. If you do not use the `-b` flag, the entire copy operation is treated as a single batch.

For more information on `bcp`, see the *Utility Programs* manual.

Getting information about partitions

`sp_helpartition` prints information about table partitions. For partitioned tables, it shows the number of data pages in the partition and summary information about data distribution. Issue `sp_helpartition`, giving the table name. This example shows data distribution immediately after creating a clustered index:

```
sp_helpartition sales
partitionid firstpage controlpage ptn_data_pages
-----
```

1	6601	6600	2782
2	13673	13672	2588
3	21465	21464	2754
4	29153	29152	2746
5	36737	36736	2705
6	44425	44424	2732
7	52097	52096	2708

```

      8      59865      59864      2755
      9      67721      67720      2851
    
```

(9 rows affected)

```

Partitions  Average Pages  Maximum Pages  Minimum Pages  Ratio (Max/Avg)
-----
          9           2735           2851           2588           1.042413
    
```

sp_helppartition shows how evenly data is distributed between partitions. The final column in the last row shows the ratio of the average column size to the maximum column size. This ratio is used to determine whether a query can be run in parallel. If the maximum is twice as large as the average, the optimizer does not choose a parallel plan.

Uneven distribution of data across partitions is called **partition skew**.

If a table is not partitioned, sp_helppartition prints the message “Object is not partitioned.” When used without a table name, sp_helppartition prints the names of all user tables in the database and the number of partitions for each table. sp_help calls sp_helppartition when used with a table name.

Using *bcp* to correct partition balance

If you need to load additional data into a partitioned table that does not have clustered indexes, and sp_helppartition shows that some partitions contain many more pages than others, you can use the bulk copy session to help balance number of rows on each partition.

The following example shows that the table has only 487 pages on one partition, and 917 on another:

```

partitionid  firstpage  controlpage  ptn_data_pages
-----
          1      189825      189824          812
          2      204601      204600          487
          3      189689      189688          917
    
```

(3 rows affected)

```

Partitions  Average Pages  Maximum Pages  Minimum Pages  Ratio (Max/Avg)
-----
          3           738           917           487           1.242547
    
```

The number of rows to add to each partition can be computed by:

- Determining the average number of rows that would be in each partition if they were evenly balanced, that is, the sum of the current rows and the rows to be added, divided by the number of partitions
- Estimating the current number of rows on each partition, and subtracting that from the target average

The formula can be summarized as:

$$\text{Rows to add} = (\text{total_old_rows} + \text{total_new_rows}) / \#_of_partitions - \text{rows_in_this_partition}$$

This sample procedure uses values stored in systabstats and syspartitions to perform the calculations:

```
create procedure help_skew @object_name varchar(30), @newrows int
as
declare @rows int, @pages int, @rowsperpage int,
        @num_parts int
select @rows = rowcnt, @pages = pagecnt
       from systabstats
       where id = object_id(@object_name) and indid in (0,1)
select @rowsperpage = floor(@rows/@pages)
select @num_parts = count(*) from syspartitions
       where id = object_id(@object_name)

select partitionid, (@rows + @newrows)/@num_parts -
       ptn_data_pgs(id, partitionid)*@rowsperpage as rows_to_add
       from syspartitions
       where id = object_id (@object_name)
```

Use this procedure to determine how many rows to add to each partition in the customer table, such as when 18,000 rows need to be copied in. The results are shown below the syntax.

```
help_skew customer, 18000
partitionid rows_to_add-----
           1           5255
           2           9155
           3           3995
```

Note If the partition skew is large, and the number of rows to be added is small, this procedure returns negative numbers for those rows that contain more than the average number of final rows.

Query results are more accurate if you run update statistics and update partition statistics so that table and partition statistics are current.

With the results from `help_skew`, you can then split the file containing the data to be loaded into separate files of that length, or use the `-F` (first) and `-L` (last) flags to `bcp`.

See “Using `bcp` to correct partition balance” on page 112.

Checking data distribution on devices with `sp_helpsegment`

At times, the number of data pages in a partition can be balanced, while the number of data pages on the devices in a segment becomes unbalanced.

You can check the free space on devices with `sp_helpsegment`. This portion of the `sp_helpsegment` report for the same table shown in the `sp_helppartition` example above shows that the distribution of pages on the devices remains balanced:

device	size	free_pages
-----	-----	-----
pubtune_detail01	15.0MB	4480
pubtune_detail02	15.0MB	4872
pubtune_detail03	15.0MB	4760
pubtune_detail04	15.0MB	4864
pubtune_detail05	15.0MB	4696
pubtune_detail06	15.0MB	4752
pubtune_detail07	15.0MB	4752
pubtune_detail08	15.0MB	4816
pubtune_detail09	15.0MB	4928

Effects of imbalance of data on segments and partitions

An imbalance of pages in partitions usually occurs when partitions have run out of space on the device, and extents have been allocated on another physical device. This is called **extent stealing**.

Extent stealing can take place when data is being inserted into the table with insert commands or bulk copy and while clustered indexes are being created.

The effects of an imbalance of pages in table partitions is:

- The partition statistics used by the optimizer are based on the statistics displayed by `sp_helppartition`.

As long as data distribution is balanced across the partitions, parallel query optimization will not be affected. The optimizer chooses a partition scan as long as the number of pages on the largest partition is less than twice the average number of pages per partition.

- I/O parallelism may be reduced, with additional I/Os to some of the physical devices where extent stealing placed data.
- Re-creating a clustered index may not produce the desired rebalancing across partitions when some partitions are nearly or completely full.

See “Problems when devices for partitioned tables are full” on page 128 for more information.

Determining the number of pages in a partition

You can use the `ptn_data_pgs` function or the `dbcc checktable` and `dbcc checkdb` commands to determine the number of data pages in a table’s partitions.

See the *System Administration Guide* for information about `dbcc`.

The `ptn_data_pgs` function returns the number of data pages on a partition. Its syntax is:

```
ptn_data_pgs(object_id, partition_id)
```

This example prints the number of pages in each partition of the `sales` table:

```
select partitionid,  
       ptn_data_pgs(object_id("sales"), partitionid) Pages  
from syspartitions  
where id = object_id("sales")
```

For a complete description of `ptn_data_pgs`, see the *Adaptive Server Reference Manual*.

The value returned by `ptn_data_pgs` may be inaccurate. If you suspect that the value is incorrect, run `update partition statistics`, `dbcc checktable`, `dbcc checkdb`, or `dbcc checkalloc` first, and then use `ptn_data_pgs`.

Updating partition statistics

Adaptive Server keeps statistics about the distribution of pages within a partitioned table and uses these statistics when considering whether to use a parallel scan in query processing. When you partition a table, Adaptive Server stores information about the data pages in each partition in the control page.

The statistics for a partitioned table may become inaccurate if any of the following occurs:

- The table is unpartitioned and then immediately repartitioned.
- A large number of rows are deleted.
- A large number of rows are updated, and the updates are not in-place updates.
- A large number of rows are bulk copied into some of the partitions using parallel bulk copy.
- Inserts are frequently rolled back.

If you suspect that query plans may be less than optimal due to incorrect statistics, run the update partition statistics command to update the information in the control page.

The update partition statistics command updates information about the number of pages in each partition for a partitioned table.

The update all statistics command also updates partition statistics.

Re-creating the clustered index or running reorg rebuild automatically redistributes the data within partitions and updates the partition statistics. dbcc checktable, dbcc checkdb, and dbcc checkalloc also update partition statistics as they perform checks.

Syntax for update partition statistics

Its syntax is:

```
update partition statistics table_name  
    [partition_number]
```

Use sp_helppartition to see the partition numbers for a table.

For a complete description of update partition statistics, see the *Adaptive Server Reference Manual*.

Steps for partitioning tables

You should plan the number of devices for the table's segment to balance I/O performance. For best performance, use dedicated physical disks, rather than portions of disks, as database devices, and make sure that no other objects share the devices with the partitioned table.

See the *System Administration Guide* for guidelines for creating segments.

The steps to follow for partitioning a table depends on where the table is when you start. This section provides examples for the following situations:

- The table has not been created and populated yet.
- The table exists, but it is not on the database segment where you want the table to reside.
- The table exists on the segment where you want it to reside, and you want to redistribute the data to improve performance, or you want to add devices to the segment.

Note The following sections provide procedures for a number of situations, including those in which severe space limitations in the database make partitioning and creating clustered indexes very difficult. These complex procedures are needed only in special cases. If you have ample room on your database devices, the process of partitioning and maintaining partitioned table performance requires only a few simple steps.

Backing up the database after partitioning tables

Using fast bulk copy and creating indexes in parallel both make minimally logged changes to the database, and require a full database dump.

If you change the segment mapping while you are working with partitioned tables, you should also dump the master database, since segment mapping information is stored in sysusages.

Table does not exist

To create a new partitioned table and load the data with bcp:

- 1 Create the table on the segment, using the `on segment_name` clause. For information on creating segments, see “Creating objects on segments” on page 96.
- 2 Partition the table, with one partition for each physical device in the segment.
See “alter table...partition syntax” on page 106.

Note If the input data file is not in clustered key order, and the table will occupy more than 40% of the space on the segment, and you need a clustered index.

See “Special procedures for difficult situations” on page 124.

- 3 Copy the data into the table using parallel bulk copy.
See “Using parallel bcp to copy data into partitions” on page 110 for examples using bcp.
- 4 If you do not need a clustered index, use `sp_helpartition` to verify that the data is distributed evenly on the partitions.
See “Getting information about partitions” on page 111.

If you need a clustered index, the next step depends on whether the data is already in sorted order and whether the data is well balanced on your partitions.

If the input data file is in index key order and the distribution of data across the partitions is satisfactory, you can use the `sorted_data` option and the segment name when you create the index. This combination of options runs in serial, checking the order of the keys, and simultaneously building the index tree. It does not need to copy the data into key order, so it does not perform load balancing. If you do not need referential integrity constraints, you can use `create index`.

See “Using drop index and create clustered index” on page 109.

To create a clustered index with referential integrity constraints, use `alter table...add constraint`.

See “Using constraints and alter table” on page 109.

If your data was not in index key order when it was copied in, verify that there is enough room to create the clustered index while copying the data.

Use `sp_spaceused` to see the size of the table and `sp_helpsegment` to see the size of the segment. Creating a clustered index requires approximately 120% of the space occupied by the table.

If there is not enough space, follow the steps in “If there is not enough space to re-create the clustered index” on page 121.

- 5 Create any nonclustered indexes.
- 6 Dump the database.

Table exists elsewhere in the database

If the table exists on the default segment or some other segment in the database, follow these steps to move the data to the partition and distribute it evenly:

- 1 If the table is already partitioned, but has a different number of partitions than the number of devices on the target segment, unpartition the table.
See “alter table...unpartition Syntax” on page 107.
- 2 Partition the table, matching the number of devices on the target segment.
See “alter table...partition syntax” on page 106.
- 3 If a clustered index exists, drop the index. Depending on how your index was created, use either drop index or alter table...drop constraint.
See “Using drop index and create clustered index” on page 109 or alter table...drop constraint and “Using constraints and alter table” on page 109.
- 4 Create or re-create the clustered index with the `on segment_name` clause. When the segment name is different from the current segment where the table is stored, creating the clustered index performs a parallel sort and distributes the data evenly on the partitions as it copies the rows to match the index order. This step re-creates the nonclustered indexes on the table.
See “Distributing data evenly across partitions” on page 108.
- 5 If you do not need the clustered index, you can drop it.
- 6 Dump the database.

Table exists on the segment

If the table exists on the segment, you may need to:

- Redistribute the data by re-creating a clustered index or by using bulk copy, or
- Increase the number of devices in the segment.

Redistributing data

If you need to redistribute data on partitions, your choice of method depends on how much space the data occupies on the partition. If the space the table occupies is less than 40 to 45% of the space in the segment, you can create a clustered index to redistribute the data.

If the table occupies more than 40 to 45% of the space on the segment, you need to bulk copy the data out, truncate the table, and copy the data in again. The steps you take depend on whether you need a clustered index and whether the data is already in clustered key order.

Use `sp_helpsegment` and `sp_spaceused` to see if there is room to create a clustered index on the segment.

If there is enough space to create or re-create the clustered index

If there is enough space, see “Distributing data evenly across partitions” on page 108 for the steps to follow. If you do not need the clustered index, you can drop it without affecting the data distribution.

Dump the database after creating the clustered index.

If there is not enough space on the segment, but space exists elsewhere on the server

If there is enough space for a copy of the table, you can copy the table to another location and then re-create the clustered index to copy the data back to the target segment.

The steps vary, depending on the location of the temporary storage space:

- On the default segment of the database or in `tempdb`
- On other segments in the database

Using the default segment or `tempdb`

- 1 Use `select into` to copy the table to the default segment or to `tempdb`.

```
select * into temp_sales from sales
```

or

```
select * into tempdb..temp_sales from sales
```

- 2 Drop the original table.
- 3 Partition the copy of the table.
- 4 Create the clustered index on the segment where you want the table to reside.
- 5 Use `sp_rename` to change the table's name back to the original name.
- 6 Dump the database.

Using space on another segment

If there is space available on another segment:

- 1 Create a clustered index, specifying the segment where the space exists. This moves the table to that location.
- 2 Drop the index.
- 3 Re-create the clustered index, specifying the segment where you want the data to reside.
- 4 Dump the database.

If there is not enough space to re-create the clustered index

If there is not enough space, and you need a to re-create a clustered index on the tables:

- 1 Copy out the data using bulk copy.
- 2 Unpartition the table.
See “alter table...unpartition Syntax” on page 107.
- 3 Truncate the table with `truncate table`.
- 4 Drop the clustered index using `drop index` or `alter table...drop constraint`.
Then, drop nonclustered indexes, to avoid deadlocking during the parallel bulk copy sessions.
See “Distributing data evenly across partitions” on page 108.
- 5 Repartition the table.
See “alter table...partition syntax” on page 106.

- 6 Copy the data into the table using parallel bulk copy. You must take care to copy the data to each segment in index key order, and specify the number of rows for each partition to get good distribution.

See “Using parallel bcp to copy data into partitions” on page 110.

- 7 Re-create the index using the with sorted_data and on *segment_name* clauses. This command performs a serial scan of the table and builds the index tree, but does not copy the data.

Do not specify any of the clauses that require data copying (fillfactor, ignore_dup_row, and max_rows_per_page).

- 8 Re-create any nonclustered indexes.
- 9 Dump the database.

If there is not enough space, and no clustered index is required

If there is no clustered index, and you do not need to create one:

- 1 Copy the data out using bulk copy.

- 2 Unpartition the table.

See “alter table...unpartition Syntax” on page 107.

- 3 Truncate the table with truncate table.

- 4 Drop nonclustered indexes, to avoid deadlocking during the parallel bulk copy in sessions.

- 5 Repartition the table.

See “alter table...partition syntax” on page 106.

- 6 Copy the data in using parallel bulk copy.

See “Using parallel bcp to copy data into partitions” on page 110.

- 7 Re-create any nonclustered indexes.
- 8 Dump the database.

If there is no clustered index, not enough space, and a clustered index is needed

To change index keys on the clustered index of a partitioned table, or if you want to create an index on a table that has been stored as a heap, performing an operating system level sort can speed the process.

Creating a clustered index requires 120% of the space used by the table to create a copy of the data and build the index tree.

If you have access to a sort utility at the operating system level:

- 1 Copy the data out using bulk copy.
- 2 Unpartition the table.
See “alter table...unpartition Syntax” on page 107.
- 3 Truncate the table with truncate table.
- 4 Drop nonclustered indexes, to avoid deadlocking during the parallel bulk copy in sessions.
- 5 Repartition the table.
See “alter table...partition syntax” on page 106.
- 6 Perform an operating system sort on the file.
- 7 Copy the data in using parallel bulk copy.
See “Using parallel bcp to copy data into partitions” on page 110.
- 8 Re-create the index using the sorted_data and on *segment_name* clauses. This command performs a serial scan of the table and builds the index tree, but does not copy the data.

Do not specify any of the clauses that require data copying (fillfactor, ignore_dup_row, and max_rows_per_page).
- 9 Re-create any nonclustered indexes.
- 10 Dump the database.

Adding devices to a segment

To add a device to a segment, follow these steps:

- 1 Use sp_helpsegment to check the amount of free space available on the devices in the segment with.

If space on any device is extremely low, see “Problems when devices for partitioned tables are full” on page 128.

You may need to copy the data out and back in again to get good data distribution.

- 2 Initialize each device with disk init, and make it available to the database with alter database.

- 3 Use `sp_extendsegment` *segment_name*, *device_name* to extend the segment to each device. Drop the default and system segment from each device.
- 4 Unpartition the table.
See “alter table...unpartition Syntax” on page 107.
- 5 Repartition the table, specifying the new number of devices in the segment.
See “alter table...partition syntax” on page 106.
- 6 If a clustered index exists, drop and re-create it. Do not use the `sorted_data` option.
See “Distributing data evenly across partitions” on page 108.
- 7 Dump the database.

Special procedures for difficult situations

These techniques are more complex than those presented earlier in the chapter.

Clustered indexes on large tables

To create a clustered index on a table that will fill more than 40 to 45% of the segment, and the input data file is not in order by clustered index key, these steps yield good data distribution, as long as the data that you copy in during step 6 contains a representative sample of the data.

- 1 Copy the data out.
- 2 Unpartition the table.
See “alter table...unpartition Syntax” on page 107.
- 3 Truncate the table.
- 4 Repartition the table.
See “alter table...partition syntax” on page 106.
- 5 Drop the clustered index and any nonclustered indexes. Depending on how your index was created, use either `drop index`.

See “Using drop index and create clustered index” on page 109) or alter table...drop constraint and “Using constraints and alter table” on page 109.

- 6 Use parallel bulk copy to copy in enough data to fill approximately 40% of the segment. This must be a representative sample of the values in the key column(s) of the clustered index.

Copying in 40% of the data is much more likely to yield good results than smaller amounts of data, you can perform this portion of the bulk copy can be performed in parallel; you must use nonparallel bcp for the second bulk copy operation.

See “Using parallel bcp to copy data into partitions” on page 110.

- 7 Create the clustered index on the segment, do not use the sorted_data clause.
- 8 Use nonparallel bcp, in a single session, to copy in the rest of the data. The clustered index directs the rows to the correct partitions.
- 9 Use sp_helppartition to check the distribution of data pages on partitions and sp_helpsegment to check the distribution of pages on the segment.
- 10 Create any nonclustered indexes.
- 11 Dump the database.

One drawback of this method is that once the clustered index exists, the second bulk copy operation can cause page splitting on the data pages, taking slightly more room in the database. However, once the clustered index exists, and all the data is loaded, future maintenance activities can use simpler and faster methods.

Alternative for clustered indexes

This set of steps may be useful when:

- The table data occupies more than 40 to 45% of the segment.
- The table data is not in clustered key order, and you need to create a clustered index.
- You do not get satisfactory results trying to load a representative sample of the data, as explained in “Clustered indexes on large tables” on page 124.

This set of steps successfully distributes the data in almost all cases, but requires careful attention:

- 1 Find the minimum value for the key column for the clustered index:

```
select min(order_id) from orders
```

- 2 If the clustered index exists, drop it. Drop any nonclustered indexes.

See “Using drop index and create clustered index” on page 109 or “Using constraints and alter table” on page 109.

- 3 Execute the command:

```
set sort_resources on
```

This command disables create index commands. Subsequent create index commands print information about how the sort will be performed, but do not create the index.

- 4 Issue the command to create the clustered index, and record the partition numbers and values in the output. This example shows the values for a table on four partitions:

```
create clustered index order_cix  
on orders(order_id)
```

The Create Index is done using Parallel Sort

Sort buffer size: 1500

Parallel degree: 25

Number of output devices: 3

Number of producer threads: 4

Number of consumer threads: 4

The distribution map contains 3 element(s) for 4 partitions.

Partition Element: 1

450977

Partition Element: 2

903269

Partition Element: 3

1356032

Number of sampled records: 2449

These values, together with the minimum value from step 1, are the key values that the sort uses as diameters when assigning rows to each partition.

- 5 Bulk copy the data out, using character mode.
- 6 Unpartition the table.

See “alter table...unpartition Syntax” on page 107.

- 7 Truncate the table.
- 8 Repartition the table.
See “alter table...partition syntax” on page 106.
- 9 In the resulting output data file, locate the minimum key value and each of the key values identified in step 4. Copy these values out to another file, and delete them from the output file.
- 10 Copy into the table, using parallel bulk copy to place them on the correct segment. For the values shown above, the file might contain:

1	Jones	...
450977	Smith	...
903269	Harris	...
1356032	Wilder	...

The bcp commands look like this:

```
bcp testdb..orders:1 in keyrows -F1 -L1
bcp testdb..orders:2 in keyrows -F2 -L2
bcp testdb..orders:3 in keyrows -F3 -L3
bcp testdb..orders:4 in keyrows -F4 -L4
```

At the end of this operation, you will have one row on the first page of each partition – the same row that creating the index would have allocated to that position.

- 11 Turn set `sort_resources` off, and create the clustered index on the segment, using the `with sorted_data` option.
Do not include any clauses that force the index creation to copy the data rows.
- 12 Use bulk copy to copy the data into the table.
Use a single, nonparallel session. You cannot specify a partition for bulk copy when the table has a clustered index, and running multiple sessions runs the risk of deadlocking.
The clustered index forces the pages to the correct partition.
- 13 Use `sp_helpartition` to check the balance of data pages on the partitions and `sp_helpsegment` to balance of pages on the segments.
- 14 Create any nonclustered indexes.

15 Dump the database.

While this method can successfully make use of nearly all of the pages in a partition, it has some disadvantages:

- The entire table must be copied by a single, slow bulk copy
- The clustered index is likely to lead to page splitting on the data pages if the table uses allpages locking, so more space might be required.

Problems when devices for partitioned tables are full

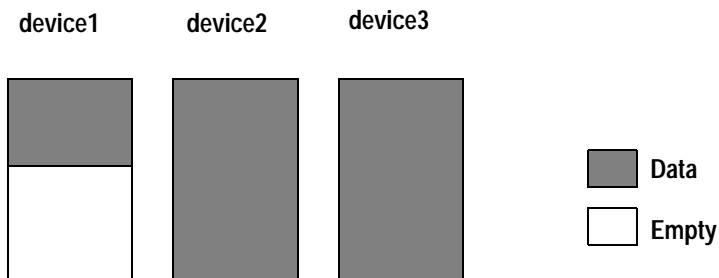
Simply adding disks and re-creating indexes when partitions are full may not solve load-balancing problems. If a physical device that underlies a partition becomes completely full, the data-copy stage of re-creating an index cannot copy data to that physical device.

If a physical device is almost completely full, re-creating 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. Figure 6-2 and Figure 6-3 show one such situation.

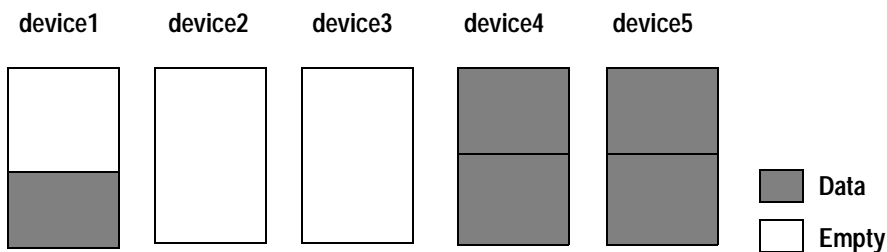
Devices 2 and 3 are completely full, as shown in Figure 6-2.

Figure 6-2: A table with 3 partitions on 3 devices

Adding two devices, repartitioning the table to use five partitions, and dropping and re-creating 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 could not be created on the device.
Devices 4 and 5	Each device has two partitions, and each is 100% full.

Figure 6-3 shows these results.

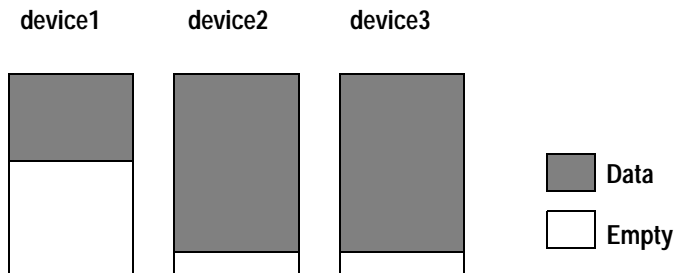
Figure 6-3: Devices and partitions after create index

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

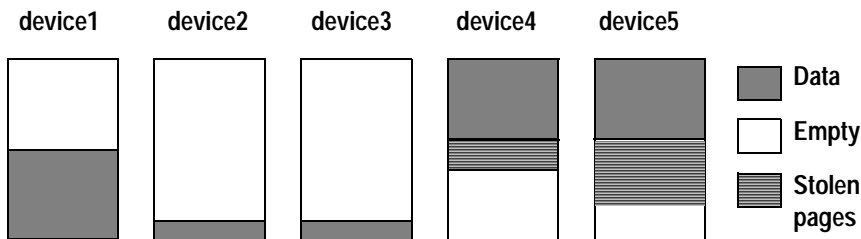
If a device is nearly full, re-creating a clustered index does not balance data across devices. Instead, the device that is nearly full stores a small portion of the partition, and the other space allocations for the partition steals extents on other devices. Figure 6-4 shows a table with nearly full data devices.

Figure 6-4: Partitions almost completely fill the devices



After adding devices and re-creating the clustered index, the result might be similar to the results shown in Figure 6-5.

Figure 6-5: 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.

In this case, a second index re-creation step might lead to a more balanced distribution. However, if one of the devices is nearly filled by extent stealing, another index creation does not solve the problem.

Using bulk copy to copy the data out and back in again is the only sure 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

Partitioned table maintenance activity requirements 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. In the second case, 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 re-create 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_helppartition` to check the balance on partitions.
If some partitions are significantly larger or smaller than the average, re-create the clustered index to redistribute data.
- Use `sp_helpsegment` to check the balance of space on underlying disks.
- If you re-create 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 `sp_helpdb` for free kilobytes.

In addition, run update partition statistics, if partitioned tables undergo the types of activities described in “Updating partition statistics” on page 115.

You might need to re-create 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 read for many queries.

Database Design

This covers some basic information on database design that database administrators and designers would find useful as a resource. It also covers the Normal Forms for database normalization and denormalization.

There are some major database design concepts and other tips in moving from the logical database design to the physical design for Adaptive Server.

Topic	Page
Basic design	133
Normalization	135
Denormalizing for performance	141

Basic design

Database design is the process of moving from real-world business models and requirements to a database model that meets these requirements.

Normalization in a relational database, is an approach to structuring information in order to avoid redundancy and inconsistency and to promote efficient maintenance, storage, and updating. Several “rules” or levels of normalization are accepted, each a refinement of the preceding one.

Of these, three forms are commonly used: first normal, second normal, and third normal. First normal forms, the least structured, are groups of records in which each field (column) contains unique and nonrepeating information. Second and third normal forms break down first normal forms, separating them into different tables by defining successively finer interrelationships between fields.

For relational databases such as Adaptive Server, the standard design creates tables in Third Normal Form.

When you translate an Entity-Relationship model in Third Normal Form (3NF) to a relational model:

- Relations become tables.
- Attributes become columns.
- Relationships become data references (primary and foreign key references).

Physical database design for Adaptive Server

Based on access requirements and constraints, implement your physical database design as follows:

- Denormalize where appropriate
- Partition tables where appropriate
- Group tables into databases where appropriate
- Determine use of segments
- Determine use of devices
- Implement referential integrity of constraints

Logical Page Sizes

Adaptive Server does 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.

You have to exercise caution when setting the page sizes.

There are hazards in using larger devices on a 2Gb-limit platform. If you attempt to configure a logical device larger than 2Gb where Adaptive Server does not support large devices, you may experience the following problems:

- Data corruption on databases (some releases give no error message).
- Inability to dump or load data from the database

Number of columns and column size

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

- 1024 for fixed-length columns in both all-pages-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 any 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.

Table 7-1: Limits for number of logins, users, and groups

Item	Version 12.0 limit	Version 12.5 limit	New range
Number of logins per server (SUID)	64K	2 billion plus 32K	-32768 to 2 billion
Number of users per database	48K	2 billion less 1032193	-32768 to 16383; 1048577 to 2 Billion
Number of groups per database	16K	1032193	16384 to 1048576

Normalization

When a table is normalized, the non-key columns depend on the key used.

From a relational model point of view, it is standard to have tables that are in Third Normal Form. Normalized physical design provides the greatest ease of maintenance, and databases in this form are clearly understood by developers.

However, a fully normalized design may not always yield the best performance. Sybase recommends that you design databases for Third Normal Form, however, if performance issues arise, you may have to denormalize to solve them.

Levels of normalization

Each level of normalization relies on the previous level. For example, to conform to Second Normal Form, entities must be in first Normal Form.

You may have to look closely at the tables within a database to verify if the database is normalized. You may have to change the way the normalization was done by going through a denormalization on given data before you can apply a different setup for normalization.

Use the following information to verify whether or not a database was normalized, and then use it to set up the Normal Forms you may want to use.

Benefits of normalization

Normalization produces smaller tables with smaller rows:

- More rows per page (less logical I/O)
- More rows per I/O (more efficient)
- More rows fit in cache (less physical I/O)

The benefits of normalization include:

- Searching, sorting, and creating indexes is faster, since tables are narrower, and more rows fit on a data page.
- You usually have more tables.

You can have more clustered indexes (one per table), so you get more flexibility in tuning queries.

- Index searching is often faster, since indexes tend to be narrower and shorter.
- More tables allow better use of segments to control physical placement of data.
- You usually have fewer indexes per table, so data modification commands are faster.

- Fewer null values and less redundant data, making your database more compact.
- Triggers execute more quickly if you are not maintaining redundant data.
- Data modification anomalies are reduced.
- Normalization is conceptually cleaner and easier to maintain and change as your needs change.

While fully normalized databases require more joins, joins are generally very fast if indexes are available on the join columns.

Adaptive Server is optimized to keep higher levels of the index in cache, so each join performs only one or two physical I/Os for each matching row.

The cost of finding rows already in the data cache is extremely low.

First Normal Form

The rules for First Normal Form are:

- Every column must be atomic. It cannot be decomposed into two or more subcolumns.
- You cannot have multivalued columns or repeating groups.
- Each row and column position can have only one value.


The table in Figure 7-1 violates First Normal Form, since the dept_no column contains a repeating group:

Figure 7-1: A table that violates first Normal Form

Employee (emp_num, emp_lname, dept_no)

Employee

emp_num	emp_lname	dept_no
10052	Jones	A10 C66
10101	Sims	D60

 Repeating

Normalization creates two tables and moves dept_no to the second table:

Figure 7-2: Correcting First Normal Form violations by creating two tables

Employee (emp_num, emp_lname)

Emp_dept (emp_num, dept_no)

Employee

emp_num	emp_lname
10052	Jones
10101	Sims

Emp_dept

emp_num	dept_no
10052	A10
10052	C66
10101	D60

Second Normal Form

For a table to be in Second Normal Form, every non-key field must depend on the entire primary key, not on part of a composite primary key. If a database has only single-field primary keys, it is automatically in Second Normal Form.

In Figure 7-3, the primary key is a composite key on emp_num and dept_no. But the value of dept_name depends only on dept_no, not on the entire primary key.

Figure 7-3: A table that violates Second Normal Form

Emp_dept (emp_num, dept_no, dept_name)

Emp_dept

emp_num	dept_no	dept_name
10052	A10	accounting
10074	A10	accounting
10074	D60	development

Depends on part of primary

Primary key

To normalize this table, move dept_name to a second table, as shown in Figure 7-4.

Figure 7-4: Correcting Second Normal Form violations by creating two tables

Emp_dept (emp_num, dept_no)

Dept (dept_no, dept_name)

Emp_dept

emp_num	dept_no
10052	A10
10074	A10
10074	D60

Primary

Dept

dept_no	dept_name
A10	accounting
D60	development

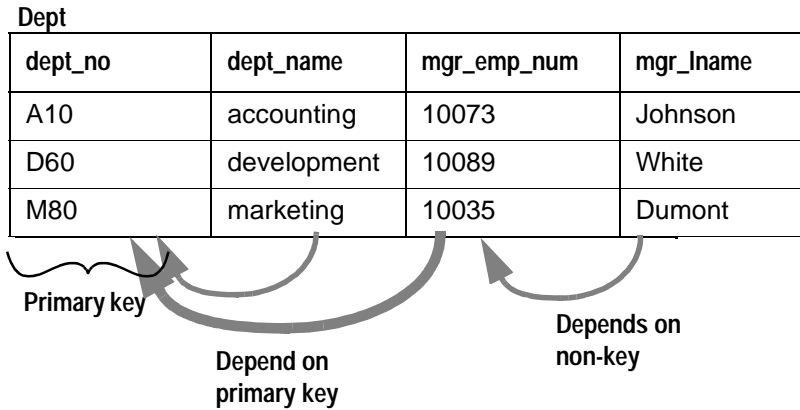
Primary

Third Normal Form

For a table to be in Third Normal Form, a non-key field cannot depend on another non-key field.

The table in Figure 7-5 violates Third Normal Form because the mgr_lname field depends on the mgr_emp_num field, which is not a key field.

Figure 7-5: A table that violates Third Normal Form
 Dept (dept_no, dept_name, mgr_emp_num, mgr_lname)



The solution is to split the Dept table into two tables, as shown in Figure 7-6. In this case, the Employees table, already stores this information, so removing the mgr_lname field from Dept brings the table into Third Normal Form.

Figure 7-6: Correcting Third Normal Form violations by creating two tables

Dept (dept_no, dept_name, mgr_emp_num)

Dept

dept_no	dept_name	mgr_emp_num
A10	accounting	10073
D60	development	10089
M80	marketing	10035

Primary

Employee (emp_num, emp_lname)

Employee

emp_num	emp_lname
10073	Johnson
10089	White
10035	Dumont

Primary

Denormalizing for performance

Once you have normalized your database, you can run benchmark tests to verify performance. You may have to denormalize for specific queries and/or applications.

Denormalizing:

- Can be done with tables or columns
- Assumes prior normalization
- Requires a thorough knowledge of how the data is being used

You may want to denormalize if:

- All or nearly all of the most frequent queries require access to the full set of joined data.

- A majority of applications perform table scans when joining tables.
- Computational complexity of derived columns requires temporary tables or excessively complex queries.

Risks

To denormalize you should have a thorough knowledge of the application. Additionally, you should denormalize only if performance issues indicate that it is needed.

For example, the `ytd_sales` column in the `titles` table of the `pubs2` database is a denormalized column that is maintained by a trigger on the `salesdetail` table. You can obtain the same values using this query:

```
select title_id, sum(qty)
       from salesdetail
       group by title_id
```

Obtaining the summary values and the document title requires a join with the `titles` table:

```
select title, sum(qty)
       from titles t, salesdetail sd
       where t.title_id = sd.title_id
       group by title
```

If you run this query frequently, it makes sense to denormalize this table. But there is a price to pay: you must create an insert/update/delete trigger on the `salesdetail` table to maintain the aggregate values in the `titles` table.

Executing the trigger and performing the changes to `titles` adds processing cost to each data modification of the `qty` column value in `salesdetail`.

This situation is a good example of the tension between decision support applications, which frequently need summaries of large amounts of data, and transaction processing applications, which perform discrete data modifications.

Denormalization usually favors one form of processing at a cost to others.

Any form of denormalization has the potential for data integrity problems that you must document carefully and address in application design.

Disadvantages

Denormalization has these disadvantages:

- It usually speeds retrieval but can slow data modification.
- It is always application-specific and must be reevaluated if the application changes.
- It can increase the size of tables.
- In some instances, it simplifies coding; in others, it makes coding more complex.

Performance advantages

Denormalization can improve performance by:

- Minimizing the need for joins
- Reducing the number of foreign keys on tables
- Reducing the number of indexes, saving storage space, and reducing data modification time
- Precomputing aggregate values, that is, computing them at data modification time rather than at select time
- Reducing the number of tables (in some cases)

Denormalization input

When deciding whether to denormalize, you need to analyze the data access requirements of the applications in your environment and their actual performance characteristics.

Often, good indexing and other solutions solve many performance problems rather than denormalizing.

Some of the issues to examine when considering denormalization include:

- What are the critical transactions, and what is the expected response time?
- How often are the transactions executed?
- What tables or columns do the critical transactions use? How many rows do they access each time?
- What is the mix of transaction types: select, insert, update, and delete?

- What is the usual sort order?
- What are the concurrency expectations?
- How big are the most frequently accessed tables?
- Do any processes compute summaries?
- Where is the data physically located?

Techniques

The most prevalent denormalization techniques are:

- Adding redundant columns
- Adding derived columns
- Collapsing tables

In addition, you can duplicate or split tables to improve performance. While these are not denormalization techniques, they achieve the same purposes and require the same safeguards.

Adding redundant columns

You can add redundant columns to eliminate frequent joins.

For example, if you are performing frequent joins on the `titleauthor` and `authors` tables to retrieve the author's last name, you can add the `au_lname` column to `titleauthor`.

Adding redundant columns eliminates joins for many queries. The problems with this solution are that it:

- Requires maintenance of new columns. you must make changes to two tables, and possibly to many rows in one of the tables.
- Requires more disk space, since `au_lname` is duplicated.

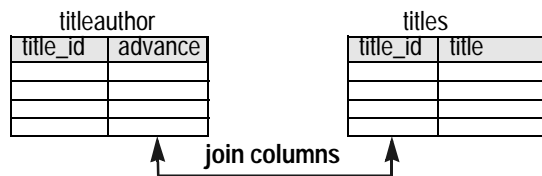
Adding derived columns

Adding derived columns can eliminate some joins and reduce the time needed to produce aggregate values. The `total_sales` column in the `titles` table of the `pubs2` database provides one example of a derived column used to reduce aggregate value processing time.

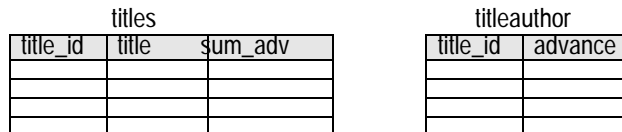
The example in Figure 7-7 shows both benefits. Frequent joins are needed between the `titleauthor` and `titles` tables to provide the total advance for a particular book title.

Figure 7-7: Denormalizing by adding derived columns

```
select title, sum(advance)
from titleauthor ta, titles t
where ta.title_id = t.title_id
group by title_id
```



```
select title, sum_adv from titles
```



You can create and maintain a derived data column in the `titles` table, eliminating both the join and the aggregate at runtime. This increases storage needs, and requires maintenance of the derived column whenever changes are made to the `titles` table.

Collapsing tables

If most users need to see the full set of joined data from two tables, collapsing the two tables into one can improve performance by eliminating the join.

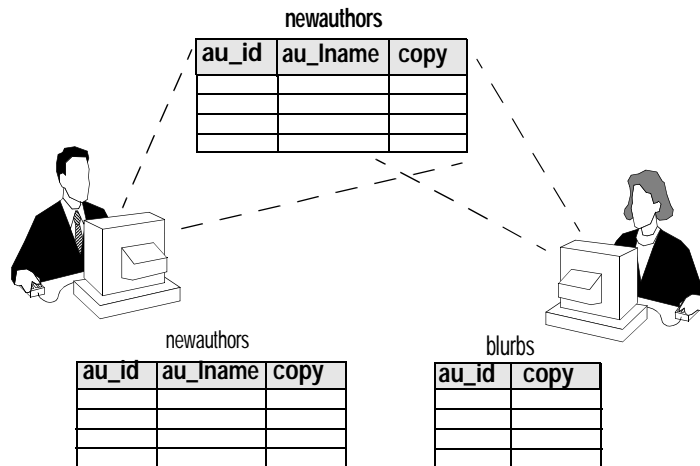
For example, users frequently need to see the author name, author ID, and the blurbs copy data at the same time. The solution is to collapse the two tables into one. The data from the two tables must be in a one-to-one relationship to collapse tables.

Collapsing the tables eliminates the join, but loses the conceptual separation of the data. If some users still need access to just the pairs of data from the two tables, this access can be restored by using queries that select only the needed columns or by using views.

Duplicating tables

If a group of users regularly needs only a subset of data, you can duplicate the critical table subset for that group.

Figure 7-8: Denormalizing by duplicating tables



The kind of split shown in Figure 7-8 minimizes contention, but requires that you manage redundancy. There may be issues of latency for the group of users who see only the copied data.

Splitting tables

Sometimes splitting normalized tables can improve performance. You can split tables in two ways:

- Horizontally, by placing rows in two separate tables, depending on data values in one or more columns
- Vertically, by placing the primary key and some columns in one table, and placing other columns and the primary key in another table

Keep in mind that splitting tables, either horizontally or vertically, adds complexity to your applications.

Horizontal splitting

Use horizontal splitting if:

- A table is large, and reducing its size reduces the number of index pages read in a query.

B-tree indexes, however, are generally very flat, and you can add large numbers of rows to a table with small index keys before the B-tree requires more levels.

An excessive number of index levels may be an issue with tables that have very large keys.

- The table split corresponds to a natural separation of the rows, such as different geographical sites or historical versus current data.

You might choose horizontal splitting if you have a table that stores huge amounts of rarely used historical data, and your applications have high performance needs for current data in the same table.

- Table splitting distributes data over the physical media, however, there are other ways to accomplish this goal.

Generally, horizontal splitting requires different table names in queries, depending on values in the tables. In most database applications this complexity usually far outweighs the advantages of table splitting .

As long as the index keys are short and indexes are used for queries on the table, doubling or tripling the number of rows in the table may increase the number of disk reads required for a query by only one index level. If many queries perform table scans, horizontal splitting may improve performance enough to be worth the extra maintenance effort.

Figure 7-9 shows how you might split the authors table to separate active and inactive authors:

Figure 7-9: Horizontal partitioning of active and inactive data

Problem: Usually only active records are accessed

Authors		
active		
active		
inactive		
active		
inactive		
inactive		

Solution: Partition horizontally into active and inactive data

Inactive_Authors		

Active_Authors		

Vertical splitting

Use vertical splitting if:

- Some columns are accessed more frequently than other columns.
- The table has wide rows, and splitting the table reduces the number of pages that need to be read.

Vertical table splitting makes even more sense when both of the above conditions are true. When a table contains very long columns that are accessed infrequently, placing them in a separate table can greatly speed the retrieval of the more frequently used columns. With shorter rows, more data rows fit on a data page, so for many queries, fewer pages can be accessed.

Managing denormalized data

Whatever denormalization techniques you use, you need to ensure data integrity by using:

- Triggers, which can update derived or duplicated data anytime the base data changes

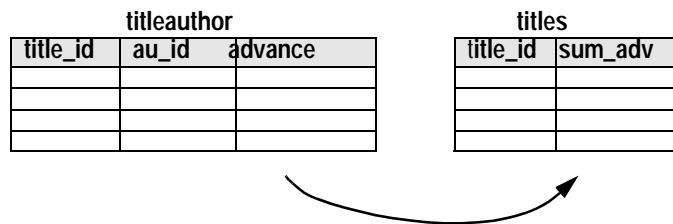
- Application logic, using transactions in each application that update denormalized data, to ensure that changes are atomic
- Batch reconciliation, run at appropriate intervals, to bring the denormalized data back into agreement

From an integrity point of view, triggers provide the best solution, although they can be costly in terms of performance.

Using triggers

In Figure 7-10, the `sum_adv` column in the `titles` table stores denormalized data. A trigger updates the `sum_adv` column whenever the `advance` column in `titleauthor` changes.

Figure 7-10: Using triggers to maintain normalized data



Using application logic

If your application has to ensure data integrity, it must ensure that the inserts, deletes, or updates to both tables occur in a single transaction.

If you use application logic, be very sure that the data integrity requirements are well documented and well known to all application developers and to those who must maintain applications.

Note Using application logic to manage denormalized data is risky. The same logic must be used and maintained in all applications that modify the data.

Batch reconciliation

If 100 percent consistency is not required at all times, you can run a batch job or stored procedure during off-hours to reconcile duplicate or derived data.

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.

It lays the foundation for understanding how to improve Adaptive Server's performance by creating indexes, tuning your queries, and addressing object storage issues.

Topic	Page
Performance gains through query optimization	151
Adaptive Server pages	153
Pages that manage space allocation	157
Space overheads	160
Heaps of data: tables without clustered indexes	167
How Adaptive Server performs I/O for heap operations	172
Caches and object bindings	174
Asynchronous prefetch and I/O on heap tables	179
Heaps: pros and cons	180
Maintaining heaps	180
Transaction log: a special heap table	181

Performance gains through query optimization

The Adaptive Server optimizer attempts to find the most efficient access path to your data for each table in the query, by estimating the cost of the physical I/O needed to access the data, and the number of times each page needs to 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:

- A 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 a non clustered index to return data, without reading the actual data rows, requiring only a fraction of the page reads required for a table scan.

Having the proper set of indexes on your tables should allow most of your 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 of your performance improvement — more than 80%, according to many performance and tuning experts — 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 useful indexes are available to help it retrieve the data. The individual 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 consume 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, along with 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 point 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 non clustered 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 12, “How Indexes Work,” describes access methods for indexes. Other chapters 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

The basic unit of storage for Adaptive Server is a **page**. Page sizes can be 2K, 4K, 8K to 16K. The server’s page size is established when you first build the source. Once the server is built the value cannot be changed. 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 version 12.5 does 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.

Adaptive Server may have to handle large volumes of data for a single query, DML operation, or command. For example, if you use a data-only-locked (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's logical pages (2K, 4K, 8K, or 16K) determines the server's space allocation. Each allocation page, object allocation map (OAM) page, data page, index page, text page, and so on are 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 object ID that the page belongs to and other information used to manage space on the page. Table 8-1 shows the number of bytes of overhead and usable space on data and index pages.

Table 8-1: 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 156.

Varying logical page sizes

Adaptive Server version 12.5 does 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 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 *Utilities Guide* for specific information about using the `dataserver` command to build your master device.

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 down the page. 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 11, “Determining Sizes of Tables and Indexes,” for more information on data and index row sizes and overhead.

The row offset table stores pointers to the starting location for each data row on the page.

Large Object (LOB) Pages

text, *image*, and Java off-row columns (LOB columns) for a table are stored as a separate data structure, consisting of a set of pages. 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 where on an 8K it is 64K, etc. 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 1392K reserved on various extents, including 48K (24 data pages) unallocated in those extents.

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.

The following types of 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 1/2MB.
- Object allocation map (OAM) pages contain information about the extents used for an object. Each table and index has at least one OAM page that tracks where pages for the object are stored in the database.
- Control pages manage space allocation for partitioned tables. Each partition has one control page.

Global allocation map pages

Each database has a Global Allocation Map Pages (GAM). It 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, its 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 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 OAM page.

Object allocation map pages

Each 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 has 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.

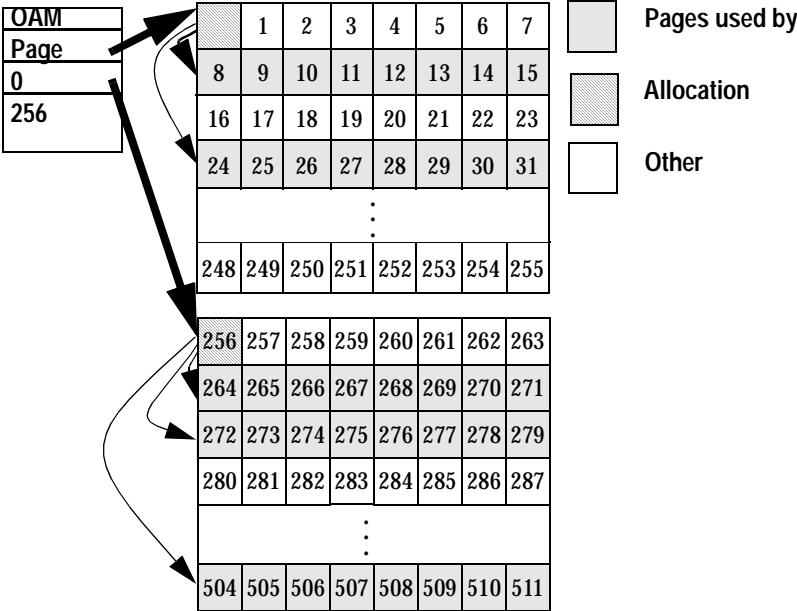
How OAM pages and allocation pages manage object storage

Figure 8-1 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 and information about the extents and pages used on the extent. So, allocation page 0 points to page 1 and 24 for the table, and allocation page 256 points to pages 272 and 504.

Figure 8-1: OAM page and allocation page pointers



Page allocation keeps an object’s pages together

Adaptive Server tries to keep the page allocations close together for objects. In most cases:

- If there is an unallocated page in the current extent, that page is assigned to the object.
- If there is no free page in the current extent, but there is an unallocated page on another of the object’s extents, that extent is used.
- If all the object’s extents are full, but there are free extents on the allocation unit, the new extent is allocated in a unit already used by the object.

sysindexes table and data access

The sysindexes table stores information about indexed and unindexed 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.

Each row in sysindexes stores pointers to a table or index to speed access to objects. Table 8-2 shows how these pointers are used during data access.

Table 8-2: Use of sysindexes 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 non clustered 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 it is configured for, 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 2K logical pages, systypes – with approximately 31 short rows, a clustered and a non-clustered index – reserves 3 extents, or 48K of memory. If you migrate the server to use 8K pages, the space reserved for systypes is still 3 extents, 192K of memory.

For a server configured for 16K, systypes requires 384K 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 all-pages-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 any 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 0-1 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 8-3: Maximum row and column length - APL & DOL

Locking scheme	Page size	Maximum row length	Maximum column length
APL tables	2K (2048 bytes)	1962	1960 bytes
	4K (4096 bytes)	4010	4008 bytes
	8K (8192 bytes)	8106	8104 bytes
	16K (16384 bytes)	16298	16296 bytes
DOL tables	2K (2048 bytes)	1964	1958 bytes
	4K (4096 bytes)	4012	4006 bytes
	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 <i>max start</i> offset of <i>varlen</i> = 8191)	8191-6-2 = 8183 bytes if table includes at least on variable length column.*
* This size includes six bytes for the row overhead and two bytes for the row length field			

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.

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:

- Two 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. 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 six bytes, plus additional overhead for the adjust table. The maximum column size, after all the overhead is taken into consideration, is less than or equal to the column length + number of bytes for adjust table + six-byte overhead.

Table 8-4: 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	8058
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. However, an insert fails if you attempt to 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:

$$\text{column length} + 10 \text{ bytes overhead}$$

Table 8-5: 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	7998
16K (16384 bytes)	16300	162290

DOL tables with variable-length columns must have an offset of less than 8191 bytes for all inserts to succeed. For example, this insert fails because the offset totals more than 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
```

The offset for columns c2, c3, and c4 is 9010, so the entire insert fails.

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. Conversion of a DOL table containing at least one maximum sized variable length column to allpages mode is restricted. Adaptive Server raises an error message and the operation is aborted.

On servers that use 16K pages, APL tables can store substantially larger sized variable length columns than can be stored in DOL tables. You can convert tables from DOL to APL, but lock scheme conversion from APL to DOL is restricted. Adaptive Server raises an error message and the operation is aborted.

Note that these restrictions on lock scheme conversions occur only if there is data in the source table that goes beyond 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. But, then, on a subsequent insert or update of the table, users might run into errors due to 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 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 8-6 displays the maximum number of datarows that can fit on a DOL data page:

Table 8-6: 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 long, the page will have 1275 bytes of unused space, after accounting for the row-offset table, and the page header.

Maximum numbers

Arguments for stored procedures

The maximum number of arguments for stored procedures is 2048. See the *Transact - SQL User's Guide* for more information.

Retrieving data with enhanced limits

Adaptive Server version 12.5 and later can store data that has different limits than data stored in previous versions. Clients also must be able to handle the new limits the data can use. 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.
- Drop and re-create the tables with wide columns.
- Insert wide data.

See the Open Client section in this guide for more information.

Heaps of data: 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*. The data rows are not stored in any particular order. This section describes how select, insert, delete, and update operations perform on heaps when there is no “useful” index to aid in retrieving data.

The phrase “no useful index” is important in describing the optimizer’s decision to perform a table scan. 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.

Other chapters in this book describe how the optimizer costs queries using indexes and how you can get more information about why the optimizer makes these choices.

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 “Index covering” on page 291.

The following sections describe how Adaptive Server locates rows when a table has no useful index.

Lock schemes and differences between heaps

The data pages in an allpages-locked table are linked into a doubly-linked 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 new pages need to be inserted, 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 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.

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 170 and “Data-only-locked heap tables” on page 172.

Select operations on heaps

When you issue a select query on a heap, and there is no useful nonclustered 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 first column in sysindexes 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 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, 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 sysindexes.root entry for the heap table stores a pointer to the last page of the heap to locate the page where the data needs to 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.

Conflicts during heap inserts

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 that lock is held until the transaction completes. If many users are trying to insert into an allpages-locked heap table at the same time, each insert must wait for the preceding transaction to complete.

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

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

Some workarounds for last-page conflicts on heaps include:

- 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

Other guidelines that apply to all transactions where there may be lock conflicts include:

- Keeping transactions short
- Avoiding 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 hint 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. When blocking occurs, 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.

If conflicts occur during heap inserts

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

- 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, and there is no useful index, Adaptive Server scans the data rows in the table to find the rows to delete. It has no way of knowing 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.

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 need to be reinserted in the same place if the transaction is rolled back.

After a delete transaction completes, 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 heaps, an update that has no useful index on the columns in the where clause performs a table scan to locate the rows that need to be changed.

Allpages-locked heap tables

Updates on allpages-locked heap tables can be performed 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 the “new” row is inserted on the last page of the table.

This type of update can cause a conflict on the last page of the heap, just as inserts do. If there are any nonclustered indexes on the table, all index references to the row need to be updated.

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.

But 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 the following steps:

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

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

If the row needs to 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 needs to read the page at the original location and then read the page where the forwarded data is stored.

Forwarded rows can be cleared from a table using the `reorg` command.

For more information on updates, see “How update operations are performed” on page 94 in the *Performance and Tuning: Optimizer* book.

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 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 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's data caches can be configured by a System Administrator to allow large I/Os. When a cache is configured to allow large I/Os, Adaptive Server can choose to 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 performs nearly eight times as fast as a single-page, 2K I/O, so queries that table scan should perform much better using large I/O.

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.

For more information on configuring memory caches for large I/O, see Chapter 10, "Memory Use and Performance."

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.

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, its I/O takes place in the default data cache. The default data cache can be configured for large I/O. If your applications include some heap tables, they will probably perform best when they use a cache configured for 16K I/O.

Heaps, 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 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.

Overview of cache strategies

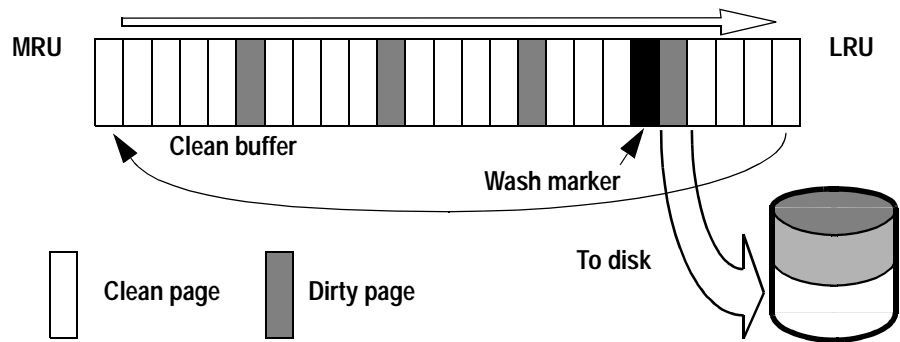
Adaptive Server has two major strategies for using its data cache efficiently:

- LRU replacement strategy, usually used for pages that a query needs to access more than once or pages that must be updated
- MRU, or *fetch-and-discard* replacement strategy, used for pages that a query needs to read only once

LRU replacement strategy

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 8-2: LRU strategy takes a clean page from the LRU end of the cache



When LRU strategy is used

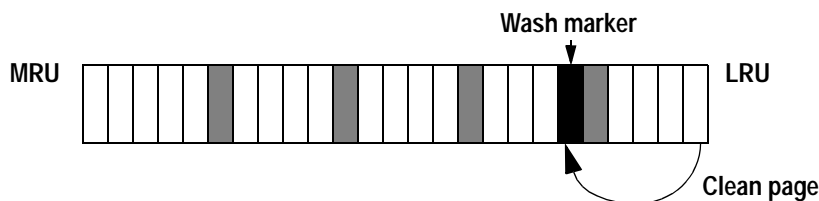
Adaptive Server uses LRU strategy for:

- 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

MRU replacement strategy

MRU (fetch-and-discard) replacement strategy is used for table scanning on heaps. This strategy places pages into the cache just before the wash marker, as shown in Figure 8-3.

Figure 8-3: MRU strategy places pages just before the wash marker



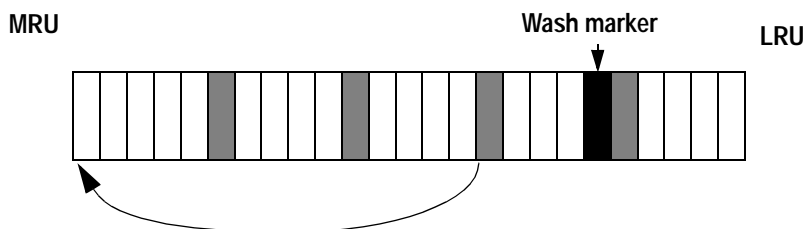
Fetch-and-discard is most often used for queries where a page is needed only once by the query. This includes:

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

Placing the pages needed only once at the wash marker means that 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 8-4: 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 heaps, select operations performing large I/O can be very effective. Adaptive Server can read sequentially through all the extents in a table.

Unless the heap is being scanned as the inner table of a nested-loop join, the 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 heaps is effective only when the page chains are not fragmented.

See “Maintaining heaps” on page 180 for information on maintaining heaps.

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 written to disk immediately.

Caching and inserts on heaps

For inserts to heap tables, the insert takes place:

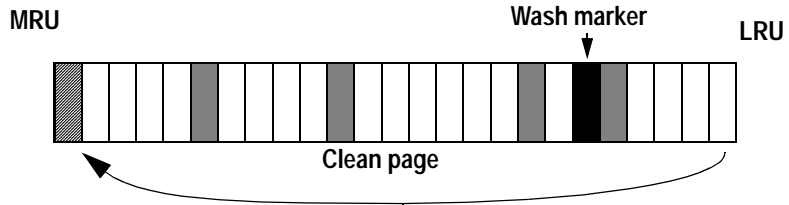
- On the last page of a table that uses allpages locking
- On a page that was recently used for a successful insert, on a table that uses data-only-locking

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 8-5. This page starts to move down the MRU/LRU chain in the data cache as other processes read pages into memory.

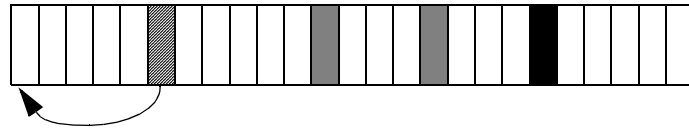
If a second insert to the page takes place while the page is still in memory, the page is located in cache, and moves back to the top of the MRU/LRU chain.

Figure 8-5: 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



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.

“Data cache” on page 215 explains more about these processes.

Caching, update and delete operations on heaps

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 then the whole 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 just before the wash marker in the cache.

Asynchronous prefetch and I/O on heap tables

Asynchronous prefetch helps speed the performance of queries that perform table scans. Any task that needs to perform a physical I/O relinquishes the server's engine (CPU) while it waits for the I/O to complete.

If a table scan needs to read 1000 pages, and none of those pages are in cache, performing 2K I/O with no asynchronous prefetch means that the task would make 1000 loops, executing on the engine, and then sleeping to wait for I/O. Using 16K I/O would required only 125 such loops.

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	Request a page. Sleep until the page has been read from disk. Wait for a turn to run on the Adaptive Server engine (CPU). Read the rows on the page.
16K I/O no prefetch	125	Request an extent. Sleep until the extent has been read from disk. Wait for a turn to run on the Adaptive Server engine (CPU). Read the rows on the 8 pages.
Prefetch	4	Request all the pages in an allocation unit. Sleep until the first page has been read from disk. Wait for a turn to run on the Adaptive Server engine (CPU). Read all the rows on all the pages in cache.

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

For more information on asynchronous prefetching, see Chapter 16, "Tuning Asynchronous Prefetch."

Heaps: pros and cons

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, having a potentially large impact in the data cache and other queries.

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

Heaps work well for small tables and tables where changes are infrequent, but they do not work well for most large tables for queries that need to return a subset of the rows.

Heaps can be useful for tables that:

- Are fairly small and use only a few pages
- Do not require direct access to a single, random row
- Do not require ordering of result sets

Partitioned heaps are useful for tables with frequent, large volumes of batch inserts where the overhead of dropping and creating clustered indexes is unacceptable. With this exception, there are very few justifications for heap tables. Most applications perform better with clustered indexes on the tables.

Maintaining heaps

Over time, I/O on heaps 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

Methods

After deletes and updates have left empty space on pages or have left empty pages on extents, use one of the following techniques 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 nonclustered indexes on the heap table. `reorg rebuild` can be used only on data-only-locked tables.

Reclaiming space by creating a clustered index

You can create and drop a clustered index on a heap table to reclaim space if updates and deletes have created many partially full pages in the table. To create a clustered index, you must have free space in the database of at least 120% of the table size.

See “Determining the space available for maintenance activities” on page 356 for more information.

Reclaiming space using *bcp*

To reclaim space with `bcp`:

- 1 Copy the table out to a file using `bcp`.
- 2 Truncate the table with the `truncate table` command.
- 3 Copy the table back in again with `bcp`.

See “Steps for partitioning tables” on page 117 for procedures for working with partitioned tables.

For more information on `bcp`, see the *Utility Guide* manual for your platform.

Transaction log: a special heap table

Adaptive Server’s 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 does not have any indexes.

Other chapters in this book describe ways to enhance the performance of the transaction log. The most important technique is to use the log on clause to create database to place your transaction log on a separate device from your data.

See the *System Administration Guide* for more information on creating databases.

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.

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.

Besides recovery, these kinds of operations require reading the transaction log:

- Any data modification that is 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 kinds of queries are still available in the data cache when Adaptive Server needs to read them, and disk I/O is not required.

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.

Topic	Page
Reducing index maintenance	183
Reducing row forwarding	189
Leaving space for forwarded rows and inserts	194
Using <code>max_rows_per_page</code> on allpages-locked tables	202

Reducing index maintenance

By default, Adaptive Server creates indexes that are completely full at the leaf level and leaves growth room for two rows on the intermediate pages.

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 use `fillfactor`, except for a `fillfactor` value of 100 percent, 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 for `create index`.

The `fillfactor` is used only when you create the index; it is not maintained over time.

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.

fillfactor values specified with `create index` are applied at the time the index is created. They are not saved in `sysindexes`, and the fullness of the data or index pages is not maintained over time.

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 185 for more information.

Advantages of using *fillfactor*

Setting fillfactor to a low value provides a temporary performance enhancement. Its benefits fade as inserts to the database increase the amount of space used on data or index pages.

A lower fillfactor provides these benefits:

- It reduces page splits on the leaf-level of indexes, and the data pages of allpages-locked tables.
- It improves data-row clustering on data-only-locked tables with clustered indexes that experience inserts.
- It 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.
- It 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 a very low fillfactor, you may notice these effects on queries and maintenance activities:

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

In some cases, it may also add a level 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.

- dbcc commands need to check more pages, so dbcc commands take more time.
- dump database time increases, because more pages need to be dumped. dump database copies all pages that store data, but does not dump pages that are not yet in use.

Your dumps and loads will take longer to complete and may use more tapes.

- Fillfactors fade away over time. If you use fillfactor to reduce the performance impact of page splits, you need to monitor your system and re-create indexes when page splitting begins to hurt performance.

Setting *fillfactor* values

sp_chgattribute allows you 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 to restore the cluster ratios of data-only-locked tables and indexes.
- Use alter table...lock to change the locking scheme for a table or you use an alter table...add/modify command that requires copying the table.
- Run create clustered index and there is a value stored for the table.

The stored fillfactor is not applied when nonclustered indexes are rebuilt 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 fill factor 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, the *fillfactor* specified in commands “create index” are applied as shown in Table 9-1.

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

Table 9-1: *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 the default fill factor percentage configuration parameter. The default *fillfactor* is applied as shown in Table 9-2.

Table 9-2: *fillfactor* values applied with 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
```

With 50 as the stored table-level value for *fillfactor*, the following create clustered index command applies the *fillfactor* values shown in Table 9-3.

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


Table 9-3: 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 a create clustered index command is run, any table-level fillfactor value stored in sysindexes is reset to 0.

To affect the filling of data-only-locked data pages during a create clustered index or reorg command, you must first issue `sp_chgattribute`.

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 a data-only-locked table is altered to use allpages locking, the fillfactor stored for the table is used for the data pages. The fillfactor stored for the clustered index is ignored.

Table 9-4 shows the fillfactors used on data and index pages by an alter table...lock command, executed after the sp_chgattribute commands above have been run.

Table 9-4: 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 9-5 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 9-5: 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

Use of the sorted_data and fillfactor options

The sorted_data option for create index is used when the data to be sorted is already in order by the index key. This allows create clustered index to skip the copy step while creating a clustered index.

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.

For more information, see “Creating an index on sorted data” on page 345.

Reducing row forwarding

Specifying an expected row size for a data-only-locked table is useful when an application allows rows that contain null values or short variable-length character fields to be inserted, and these rows grow in length with subsequent updates. The major purpose of setting an expected row size is 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 it is possible to 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 can result in row forwarding.

See “Data-only locked heap tables” on page 168 for more information.

You can 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 the table is later converted to data-only locking, the `exp_row_size` is applied during the conversion process, and to all subsequent inserts and updates.

Default, minimum, and maximum values for `exp_row_size`

Table 9-6 shows the minimum and maximum values for expected row size and the meaning of the special values 0 and 1.

Table 9-6: Valid values for expected row size

exp_row_size values	Minimum, maximum, and special values
Minimum	The greater of: <ul style="list-style-type: none"> • 2 bytes • The sum of all fixed-length columns
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 191 for information on how this parameter 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

To add or change the expected row size for a table, use `sp_chgattribute`. This sets the expected row size to 190 for the `new_titles` table:

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

If you want a table to switch to the default `exp_row_size` percent instead of a current, explicit value, 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 a clustered index on the table is created or `reorg rebuild` is run on the table. The expected row size is applied as rows are copied to new data pages.

If you increase `exp_row_size`, and re-create 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_row_size` column for all user tables in a database:

```
select object_name(id), exp_row_size
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 first 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 too much space on data pages due to over-configuring 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
```

You can also 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.

For more information on `optdiag`, see Chapter 6, “Statistics Tables and Displaying Statistics with `optdiag`.” in the *Performance and Tuning: Monitoring and Analyzing for Performance*.

Querying `systabstats` to check for forwarded rows

You can check the `forwrowcnt` column in the `systabstats` table to see the number of forwarded rows for a table. This query checks the number of forwarded rows for all user tables (those with object IDs greater than 100):

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

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

If you need to 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 9-7.

Table 9-7: Conversion of `max_rows_per_page` to `exp_row_size`

Value of <code>max_rows_per_page</code>	Value of <code>exp_row_size</code>
0	Percentage value set by default <code>exp_row_size percent</code>
1-254	The smaller of: <ul style="list-style-type: none"> • Maximum row size • $2002/\text{max_rows_per_page}$ value

For example, if `max_rows_per_page` is set to 10 for an allpages-locked table with a maximum defined row size of 300 bytes, the `exp_row_size` value will be 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 will be 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 check the number of forwarded rows still being generated by your applications. Run `reorg forwarded_rows` if you feel that the number of forwarded rows is high enough to affect application performance. `reorg forwarded_rows` uses short transactions and is very nonintrusive, so you can run it while applications are active.

See the *System Administration Guide* for more information.

If the application still results in a large number of forwarded rows, you may want to use `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 non-peak 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 needed to read a set of rows. If the increase in the number of I/Os due to increased storage space is high, then allowing rows to be forwarded and occasionally running `reorg` may have less overall performance impact.

Leaving space for forwarded rows and inserts

Setting a `reservepagegap` value can reduce the frequency of maintenance activities such as running `reorg rebuild` and re-creating indexes for some tables to maintain high performance. 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. The `reservepagegap` space management property is used 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 a clustered index is created on a table or immediately after reorg rebuild is run. However, future data modifications can cause row forwarding and can require allocation of additional data and index pages to store inserted rows.

Setting a reserve page gap can reduce storage fragmentation and reduce the frequency with which you need to re-create indexes or run reorg rebuild on the table.

Extent allocation operations and *reservepagegap*

Commands that allocate many data pages perform **extent allocation** to allocate eight pages at a time, rather than allocating just one page at a time. Extent allocation reduces logging, since it writes 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 sometimes require a table-copy operation also. All of these commands allocate an extent, and, unless a reserve page gap value is in effect, fill all eight pages.

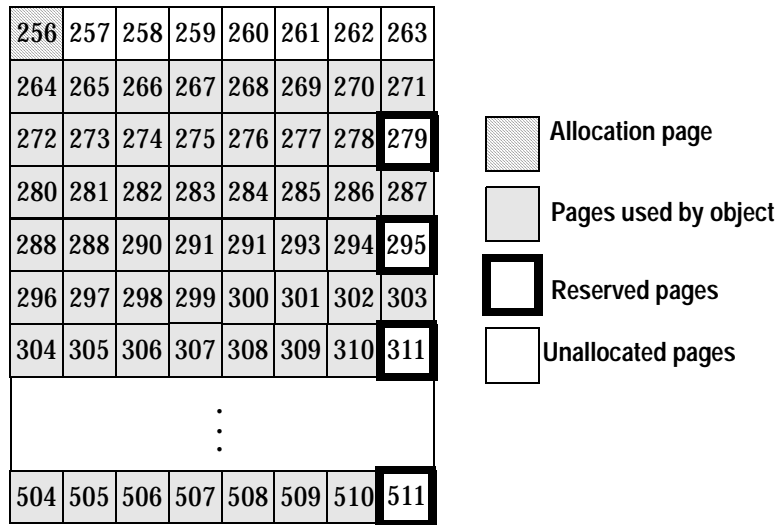
You specify the *reservepagegap* in pages, indicating a ratio of empty pages to filled pages. For example, if you specify a *reservepagegap* of 8, an operation doing extent allocation fills seven pages and leaves the eighth page empty.

These empty pages can be used to store forwarded rows and for maintaining the clustering of data rows in index key order, for data-only-locked tables with clustered indexes.

Since extent allocation operations must allocate entire extents, they 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 7 empty, unallocated pages, 248 used pages, and the allocation page. These 7 pages can be used 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.

Figure 9-1 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 9-1: 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.

The default value for `reservepagegap` is 0, meaning that no space is reserved. You can have more than 255 bytes, use pattern strings for LIKE more than 255 bytes and LIKE can also operate on wider columns.

Specifying a reserve page gap with *create index*

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

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

You can also specify a `reservepagegap` value with the `alter table...constraint` options, primary key and unique, that create indexes. This example creates a unique constraint:

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

Changing *reservepagegap*

The following command uses `sp_chgattribute` to change the reserve page gap for the titles table to 20:

```
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.
- When the `reorg rebuild` command is run on the table, the reserve page gap is applied as the table is copied to new data pages.
- When a clustered index is created, 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, while rebuilding indexes for the table
- reorg rebuild commands that affect indexes
- 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 9-8 shows the values applied when running reorg rebuild or dropping and creating a clustered index.

Table 9-8: 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)

The reservepagegap for the table is applied to both the data and index pages for an allpages-locked table with a clustered index. For a data-only-locked table, the table's 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 9-9 shows the effects of this sequence of commands.

Table 9-9: 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
- How often you re-create 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

You can use optdiag to check the cluster ratio and the number of forwarded rows in tables. Declines in cluster ratios can also indicate that running reorg commands could improve performance:

- If the data page cluster ratio for a clustered index is low, run reorg rebuild or drop and re-create the clustered index.
- If the index page cluster ratio is low, drop and re-create the non-clustered 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 6, “Statistics Tables and Displaying Statistics with `optdiag`,” in the book *Performance and Tuning: Monitoring and Analyzing for Performance* for more information on `optdiag`.

***reservepagegap* and *sorted_data* options to create index**

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.

Background on the *sorted_data* option

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 345.

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 9-10 shows how these rules apply.

Table 9-10: `reservepagegap` and `sorted_data` options

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

Matching options and goals

If you want to redistribute the data pages of a table, leaving room for later expansion:

- For allpages-locked tables, drop and re-create the clustered index without using the `sorted_data` option. Specify the desired `reservepagegap` value in the create clustered index command, if the value stored in `sysindexes` is not the value you want.
- For data-only-locked tables, use `sp_chgattribute` to set the `reservepagegap` for the table to the desired value and then drop and re-create 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 specify a `reservepagegap` with the `create clustered index` command. Alternatively, you can 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.

If you plan 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`**

`select into` does not carry over the base table's `max_rows_per_page` value, but creates the new table with a `max_rows_per_page` value of 0. Use `sp_chgattribute` to set the `max_rows_per_page` value on the target table.

Applying `max_rows_per_page` to existing data

`sp_chgattribute` configures the `max_rows_per_page` of a table or an index. `sp_chgattribute` affects all future operations; it does not change existing pages. 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
```

There are two ways to apply a `max_rows_per_page` value to existing data:

- If the table has a clustered index, drop and re-create the index with a `max_rows_per_page` value.
- Use the `bcu` utility as follows:
 - a Copy out the table data.
 - b Truncate the table.
 - c Set the `max_rows_per_page` value with `sp_chgattribute`.
 - d Copy the data back in.

Memory Use and Performance

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

Topic	Page
How memory affects performance	205
How much memory to configure	206
Caches in Adaptive Server	211
Procedure cache	212
Data cache	215
Configuring the data cache to improve performance	220
Named data cache recommendations	230
Maintaining data cache performance for large I/O	240
Speed of recovery	242
Auditing and performance	243

The *System Administration Guide* describes how to determine the best memory configuration values for Adaptive Server, and the memory needs of other server configuration options.

How memory affects performance

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

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

Memory conditions that can cause poor performance are:

- Total data cache size is too small.
- Procedure cache size is too small.
- Only the default cache is configured on an SMP system with several active CPUs, leading to contention for the data cache.
- User-configured data cache sizes are not appropriate for specific user applications.
- Configured I/O sizes are not appropriate for specific queries.
- Audit queue size is not appropriate if auditing feature is installed.

How much memory to configure

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

The total memory allocated during boot-time is the sum of memory required for all the configuration needs of Adaptive Server. This value can be obtained from the read-only configuration parameter 'total logical memory'. This value is calculated by Adaptive Server. The configuration parameter 'max memory' must be greater than or equal to 'total logical memory'. 'max memory' indicates the amount of memory you will allow for Adaptive Server needs.

During boot-time, by default, Adaptive Server allocates memory based on the value of 'total logical memory'. However, if the configuration parameter 'allocate max shared memory' has been set, then the memory allocated will be based on the value of 'max memory'. The configuration parameter 'allocate max shared memory' will enable a system administrator to allocate, the maximum memory that is allowed to be used by Adaptive Server, during boot-time.

The key points for memory configuration are:

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

- The configuration parameter 'allocate max shared memory' can be turned on during boot-time and run-time to allocate all the shared memory up to 'max memory' with the least number of shared memory segments. Large number of shared memory segments has the disadvantage of some performance degradation on certain platforms. Please check your operating system documentation to determine the optimal number of shared memory segments. Note that once a shared memory segment is allocated, it cannot be released until the next server reboot.
- Configure the different configuration parameters, if the defaults are not sufficient.
- Now the difference between 'max memory' and 'total logical memory' is additional memory available for procedure, data caches or for other configuration parameters.

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

- Adaptive Server may not start, if the physical resources on your machine does is not sufficient.
- If it does start, the operating system page fault rates may rise significantly and the operating system may need to re configured to compensate.

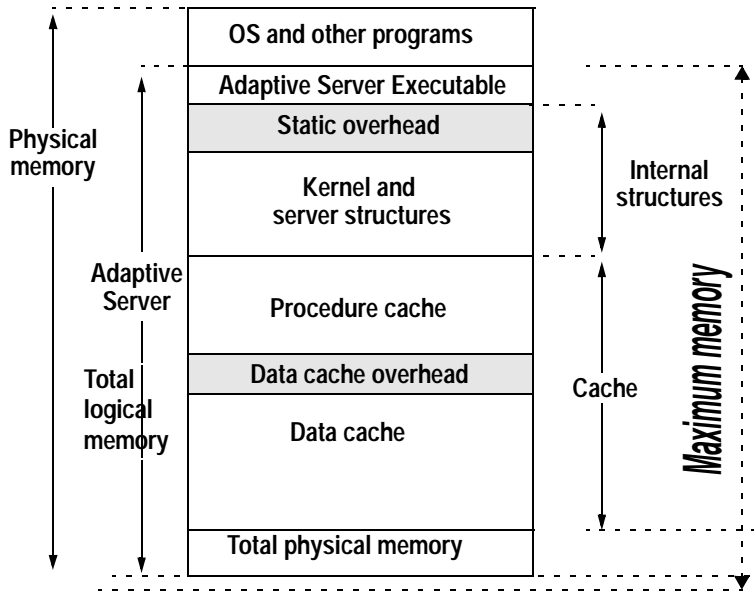
The *System Administration Guide* provides a thorough discussion of:

- How to configure the total amount of memory used by Adaptive Server
- Configurable parameters that use memory, which affects the amount of memory left for processing queries
- Handling wider character literals requires Adaptive Server to allocate memory for string user data. Also, rather than statically allocating buffers of the maximum possible size, Adaptive Server allocates memory dynamically. That is, it allocates memory for local buffers as it needs it, always allocating the maximum size for these buffers, even if large buffers are unnecessary. These memory management requests may cause Adaptive Server to have a marginal loss in performance when handling wide-character data.

- If you require Adaptive Server to handle more than 1000 columns from a single table, or process over 10000 arguments to stored procedures, the server must set up and allocate memory for various internal data structures for these objects. An increase in the number of small tasks that are performed repeatedly may cause performance degradation for queries that deal with larger numbers of such items. This performance hit increases as the number of columns and stored procedure arguments increases.
- Memory that is allocated dynamically (as opposed to rebooting Adaptive Server to allocate the memory) slightly degrades the server's performance.
- When Adaptive Server uses larger logical page sizes, all disk I/Os are done in terms of the larger logical page sizes. For example, if Adaptive Server uses an 8K logical page size, it retrieves data from the disk in 8K blocks. This should result in an increased I/O throughput, although the amount of throughput is eventually limited by the controller's I/O bandwidth.

What remains after all other memory needs have been met is available for the procedure cache and the data cache. Figure 10-1 shows how memory is divided.

Figure 10-1: How Adaptive Server uses memory



Dynamic reconfiguration

Dynamic memory allocation

Adaptive Server allows you to allocate total physical memory dynamically. Many of the configuration parameters that consume memory were static in pre-12.5 versions of Adaptive Server, and the server needed to be restarted when more memory was required. For example, when you changed the number of user connections, you had to restart the server for this to take effect. Many of the configuration parameter that effect memory are now dynamic, and the server does not have to be restarted for them to take effect. For a full list of the configuration parameters that have changed from static to dynamic, see Table 10-1.

Table 10-1: Dynamic configuration parameters

Configuration parameter	Configuration parameter
addition network memory	number of pre-allocated extents
audit queue size	number of user connections
cpu grace time	number of worker processes
deadlock pipe max messages	open index hash spinlock ratio
default database size	open index spinlock ratio
default fill factor percent	open object spinlock ratio
disk i/o structures	partition groups
errorlog pipe max messages	partition spinlock ratio
max cis remote connections	permission cache entries
memory per worker process	plan text pipe max messages
number of alarms	print recovery information
number of aux scan descriptors	process wait events
number of devices	size of global fixed heap
number of dtx participants	size of process object heap
number of java sockets	size of shared class heap
number of large i/o buffers	size of unilib cache
number of locks	sql text pipe max messages
number of mailboxes	statement pipe max messages
number of messages	tape retention in days
number of open databases	time slice
number of open indexes	user log cache spinlock ratio
number of open objects	

How memory is allocated

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

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

If you upgrade to release 12.5 Adaptive Server or higher, pre-12.5 Adaptive Server configuration values are used to calculate the new values for the procedure cache size. Adaptive Server computes the size of the default data cache during the upgrade and writes this value to the configuration file. If the computed sizes of the data cache or procedure cache are less than the default sizes, they are reset to the default. During the upgrade, max memory is set to the value of total logical memory specified in the configuration file.

Caches in Adaptive Server

Once the procedure cache and the data cache are configured there is no division or left over memory.

- The **procedure cache** – used for stored procedures and triggers and for short-term memory needs such as statistics and query plans for parallel queries.
- The **data cache** – used for data, index, and log pages. The data cache can be divided into separate, named caches, with specific databases or database objects bound to specific caches.

Set the procedure cache size to an absolute value using `sp_configure`. See the *System Administration Guide* for more information.

CAche sizes and buffer pools

Memory page sizes are in multiples of 2K (i.e. max memory, total logical memory, and so on), procedure cache is in terms of 2K pages. Buffer cache is in terms of logical page size units.

Large I/O is scaled in terms of an extent I/O. This means that with an 8K logical page size, a large I/O means a 64k read/write.

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

You have to be careful in how you set up the logical page sizes and what you allow for in the buffer pool sizes.

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

Procedure cache

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

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

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

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

When you first install Adaptive Server, the default procedure cache size is 3271 memory pages. The optimum value for the procedure cache varies from application to application, and it may also vary as usage patterns change. The configuration parameter to set the size, procedure cache size, is documented in the *System Administration Guide*.

Getting information about the procedure cache size

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

proc buffers

Represents the maximum number of compiled procedural objects that can reside in the procedure cache at one time.

proc headers

Represents the number of pages dedicated to the procedure cache. Each object in cache requires at least 1 page.

Monitoring procedure cache performance

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

For more information, see “Procedure cache management” on page 96 in the book *Performance and Tuning: Monitoring and Analyzing for Performance*.

Procedure cache errors

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

Procedure cache sizing

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

The formulas in Figure 10-2 suggest a good starting point.

Figure 10-2: Formulas for sizing the procedure cache

$$\text{Procedure cache size} = \frac{(\text{Max \# of concurrent users}) * (4 + \text{Size of largest plan}) * 1.25}{1}$$

$$\text{Minimum procedure cache size needed} = \frac{(\text{\# of main procedures}) * (\text{Average plan size})}{1}$$

If you have nested stored procedures (for example, A, B and C)—procedure A calls procedure B, which calls procedure C—all of them need to be in the cache at the same time. Add the sizes for nested procedures, and use the largest sum in place of “Size of largest plan” in the formula in Figure 10-2.

The minimum procedure cache size is the smallest amount of memory that allows at least one copy of each frequently used compiled object to reside in cache. However, the procedure cache can also be used as additional memory at execution time, such as when an ad hoc query uses the distinct keyword which uses the internal lmlink function that will dynamically allocate memory from the procedure cache. Then the create index will also use the procedure cache memory and can generate the 701 error though no stored procedure is involved.

For additional information on sizing the procedure cache see “Using sp_monitor to measure CPU usage” on page 53.

Estimating stored procedure size

To get a rough estimate of the size of a single stored procedure, view, or trigger, use:

```
select(count(*) / 8) +1
       from sysprocedures
       where id = object_id("procedure_name")
```

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

```
select(count(*) / 8) +1
       from sysprocedures
       where id = object_id("titleid_proc")
```

```
-----
3
```

Data cache

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

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

Default cache at installation time

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

The following pages describe the way this single data cache is used. “Configuring the data cache to improve performance” on page 220 describes how to improve performance by dividing the data cache into named caches and how to bind particular objects to these named caches.

Most of the concepts on aging, buffer washing, and caching strategies apply to both the user-defined data caches and the default data cache.

Page aging in data cache

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

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

Typically, pages in the wash section are clean, i.e. the I/O on these pages have been completed. When a task or query wants to grab a page from LRU end it expects the page to be clean. If not, the query has to wait for the I/O to complete on the page before it can be grabbed which impairs performance.

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

See the *System Administration Guide* for more information.

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

When the checkpoint process estimates that the number of changes to a database will take longer to recover than the configured value of the recovery interval configuration parameter, it traverses the cache, writing dirty pages to disk.

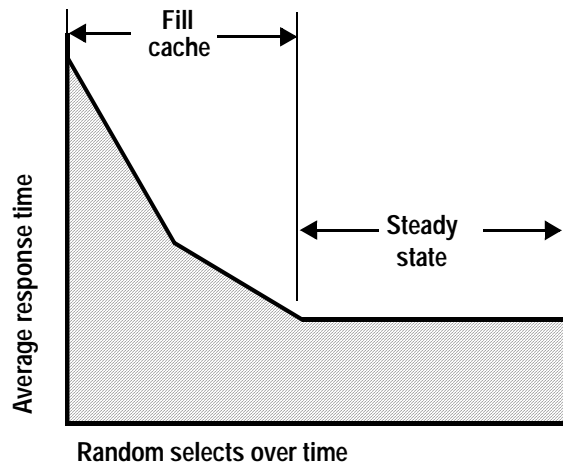
- Recovery uses only the default data cache making it faster.
- The housekeeper wash task writes dirty pages to disk when idle time is available between user processes.

Effect of data cache on retrievals

Figure 10-3 shows the effect of data caching on a series of random select statements that are executed over a period of time. If the cache is empty initially, the first select statement is guaranteed to require disk I/O. You have to be sure to adequately size the data cache for the number of transactions you expect against the database.

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

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

Figure 10-3: Effects of random selects on the data cache

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

Effect of data modifications on the cache

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

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

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

Data cache performance

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

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

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

Testing data cache performance

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

- Empty
- Fully randomized
- Partially randomized
- Deterministic

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

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

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

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

Cache hit ratio for a single query

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

To compute the cache hit ratio, use this formula:

Figure 10-4:

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

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

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

Cache hit ratio information from `sp_sysmon`

`sp_sysmon` reports on cache hits and misses for:

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

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

See “Cache statistics summary (all caches)” on page 84 in the book *Performance and Tuning: Monitoring and Analyzing for Performance*.

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

See “Cache management by cache” on page 89 in the book *Performance and Tuning: Monitoring and Analyzing for Performance*.

Configuring the data cache to improve performance

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

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

- 1 To reduce contention on the default data cache spinlock, divide the cache into n where n is 1, 2, 4, 8, 16, 32 or 64. If you have contention on the spinlock with 1 cache partition, the contention is expected to reduce x/n where n is the number of partitions.
- 2 When a particular cache partition spinlock is *hot*, consider splitting the default cache into named caches.
- 3 If there is still contention, consider splitting the named cache into named cache partitions.

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

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

Configuring the data cache can improve performance in the following ways:

- You can configure named data caches large enough to hold critical tables and indexes.

This keeps other server activity from contending for cache space and speeds up queries using these tables, since the needed pages are always found in cache.

You can configure these caches to use relaxed LRU replacement policy, which reduces the cache overhead.

- You can bind a “hot” table—a table in high demand by user applications—to one cache and the indexes on the table to other caches to increase concurrency.
- You can create a named data cache large enough to hold the “hot pages” of a table where a high percentage of the queries reference only a portion of the table.

For example, if a table contains data for a year, but 75% of the queries reference data from the most recent month (about 8% of the table), configuring a cache of about 10% of the table size provides room to keep the most frequently used pages in cache and leaves some space for the less frequently used pages.

- You can assign tables or databases used in decision support systems (DSS) to specific caches with large I/O configured.

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

- You can bind tempdb to its own cache to keep it from contending with other user processes.

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

- Text pages can be bound to named caches to improve the performance on text access.
- You can bind a database's log to a cache, again reducing contention for cache space and access to the cache.
- When changes are made to a cache by a user process, a **spinlock** denies all other processes access to the cache.

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

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

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

Commands to configure named data caches

The commands used to configure caches and pools are shown in Table 10-2

Table 10-2: Commands used to configure caches

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

For a full description of configuring named caches and binding objects to caches, see the *System Administration Guide*. Only a System Administrator can configure named caches and bind database objects to them.

Tuning named caches

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

- A cache that is poorly used hurts performance.
If you allocate 25% of your data cache to a database that services a very small percentage of the query activity on your server, I/O increases in other caches.
- A pool that is unused hurts performance.
If you add a 16K pool, but none of your queries use it, you have taken space away from the 2K pool. The 2K pool's cache hit ratio is reduced, and I/O is increased.
- A pool that is overused hurts performance.

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

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

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

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

Cache configuration goals

Goals for configuring caches are:

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

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

Gather data, plan, and then implement

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

- Determine the maximum amount of memory you can allocate to Adaptive Server. Set 'max memory' configuration parameter to that value.
- Once all the configuration parameters that use Adaptive Server memory have been configured, the difference between the 'max memory' and run value of 'total logical memory' is the memory available for additional configuration and/or for data/procedure caches. If you have sufficiently configured all the other configuration parameters, you can choose to allocate this additional memory to data caches. Most changes to the data cache are dynamic and do not require a reboot.
- Note that if you allocate all the additional memory to data caches, there may not be any memory available for reconfiguration of other configuration parameters. However, if there is additional memory available in your system, 'max memory' value can be increased dynamically and other dynamic configuration parameters like 'procedure cache size', 'user connections, etc., can be increased.
- Use your performance monitoring tools to establish baseline performance, and to establish your tuning goals.

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

Decide the data caches's size by looking at existing objects and applications. Note that addition of new caches or increase in configuration parameters that consume memory does not reduce the size of the default data cache. Once you have decided the memory available for data caches and size of each individual cache, add new caches and increase or decrease size of existing data caches.

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

- Once these sizes are determined, examine remaining I/O patterns, cache contention, and query performance. Configure caches proportional to I/O usage for objects and databases.

Keep your performance goals in mind as you configure caches:

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

Moving a few high-I/O objects to separate caches also reduces the spinlock contention and improves performance.

- If your major goal is to improve response time by improving cache hit ratios for particular queries or applications, creating caches for the tables and indexes used by those queries should be guided by a thorough understanding of the access methods and I/O requirements.

Evaluating cache needs

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

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

See “Disk I/O management” on page 102 in the book *Performance and Tuning: Monitoring and Analyzing for Performance* for information about the `sp_sysmon` output showing disk I/O.

Large I/O and performance

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

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

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

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

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

- Queries that scan entire tables. For example:

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

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

```
where indexed_colname >= value
```

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

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

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

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

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

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

Note Adaptive Server version 12.5.03 or later enables large-page allocation in `select into`. It allocates pages by extent rather than by individual page, thus issuing fewer logging requests for the target table.

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

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

The optimizer and cache choices

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

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

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

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

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

Reducing spinlock contention with cache partitions

As the number of engines and tasks running on an SMP system increases, contention for the spinlock on the data cache can also increase. Any time a task needs to access the cache to find a page in cache or to relink a page on the LRU/MRU chain, it holds the cache spinlock to prevent other tasks from modifying the cache at the same time.

With multiple engines and users, tasks wind up waiting for access to the cache. Adding cache partitions separates the cache into partitions that are each protected by its own spinlock. When a page needs to be read into cache or located, a hash function is applied to the database ID and page ID to identify which partition holds the page.

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

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

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

For more information on configuring cache partitions, see the *System Administration Guide*.

For information on monitoring cache spinlock contention with `sp_sysmon`, see “Cache spinlock contention” on page 89 in the book *Performance and Tuning: Monitoring and Analyzing for Performance*.

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

Cache replacement strategies and policies

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

Strategies

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

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

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

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

See “Specifying the cache strategy” on page 45 and “Controlling large I/O and cache strategies” on page 47 in the book *Performance and Tuning: Optimizer* for information on specifying the cache strategy in queries or setting values for tables.

Policies

A System Administrator can specify whether a cache is going to be maintained as an MRU/LRU-linked list of pages (*strict*) or whether *relaxed LRU replacement policy* can be used. The two replacement policies are:

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

The default cache replacement policy is strict replacement. Relaxed replacement policy should be used only when both of these conditions are true:

- There is little or no replacement of buffers in the cache.
- The data is not updated or is updated infrequently.

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

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

See the *System Administration Guide* for more information.

Configuring relaxed LRU Replacement for database logs

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

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

Relaxed LRU replacement for lookup tables and indexes

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

Named data cache recommendations

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

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

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

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

See the *System Administration Guide* for specific information.

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

See “Choosing the I/O size for the transaction log” on page 234.

- If a table or its index is small and constantly in use, create a cache for just that object or for a few objects.
- For caches with cache hit ratios of more than 95%, configure relaxed LRU cache replacement policy if you are using multiple engines.
- Keep cache sizes and pool sizes proportional to the cache utilization objects and queries:
 - If 75% of the work on your server is performed in one database, that database should be allocated approximately 75% of the data cache, in a cache created specifically for the database, in caches created for its busiest tables and indexes, or in the default data cache.
 - If approximately 50% of the work in your database can use large I/O, configure about 50% of the cache in a 16K memory pool.

- It is better to view the cache as a shared resource than to try to micromanage the caching needs of every table and index.

Start cache analysis and testing at the database level, concentrating on particular tables and objects with high I/O needs or high application priorities and those with special uses, such as tempdb and transaction logs.

- On SMP servers, use multiple caches to avoid contention for the cache spinlock:

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

Use `sp_sysmon` periodically during high-usage periods to check for cache contention.

See “Cache spinlock contention” on page 89 in the book *Performance and Tuning: Monitoring and Analyzing for Performance*.

- Set relaxed LRU cache policy on caches with cache hit ratios of more than 95%, such as those configured to hold an entire table or index.

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

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

Determining cache sizes for special tables or indexes

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

```
sp_spaceused table_name, 1
```

Examining cache needs for *tempdb*

Look at your use of *tempdb*:

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

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

These queries can use 16K I/O, so you can use this information to help you size a 16K pool for the *tempdb* cache.

- Estimate the duration (in wall-clock time) of the temporary tables and worktables.
- Estimate how often queries that create temporary tables and worktables are executed.

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

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

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

Examining cache needs for transaction logs

On SMP systems with high transaction rates, binding the transaction log to its own cache can greatly reduce cache spinlock contention in the default data cache. In many cases, the log cache can be very small.

The current page of the transaction log is written to disk when transactions commit, so your objective in sizing the cache or pool for the transaction log is not to avoid writes. Instead, you should try to size the log to reduce the number of times that processes that need to reread log pages must go to disk because the pages have been flushed from the cache.

Adaptive Server processes that need to read log pages are:

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

When sizing a cache for a transaction log:

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

Estimate how long the longest triggers and deferred updates last.

If some of your long-running transactions are rolled back, check the length of time they ran.

- Estimate the rate of growth of the log during this time period.

You can check your transaction log size with `sp_spaceused` at regular intervals to estimate how fast the log grows.

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

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

Choosing the I/O size for the transaction log

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

- When a transaction ends
- When the user log cache is full
- When the transaction changes tables in another database
- When another process needs to write a page referenced in the user log cache
- At certain system events

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

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

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

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

- Your database is using 2K log I/O.
- The number of log writes per second is high.
- The average number of writes per log page is slightly above one.

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

	per sec	per xact	count	% of total
Transaction Log Writes	22.5	458.0	1374	n/a
Transaction Log Alloc	20.8	423.0	1269	n/a
Avg # Writes per Log Page	n/a	n/a	1.08274	n/a

See “Transaction log writes” on page 59 in the book *Performance and Tuning: Monitoring and Analyzing for Performance* for more information.

Configuring for large log I/O size

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

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

```
sp_logiosize "all"
```

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

```
sp_logiosize "default"
```

By default, Adaptive Server sets the log I/O size for user databases to 4K. If no 4K pool is available in the cache used by the log, 2K I/O is used instead.

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

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

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

Additional tuning tips for log caches

For further tuning after configuring a cache for the log, check `sp_sysmon` output. Look at the output for:

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

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

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

Basing data pool sizes on query plans and I/O

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

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

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

Making a good choice about pool sizes requires:

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

Checking I/O size for queries

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

Logical page size	Memory pool
2K	16K
4K	64K
8K	128K
16K	256K

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

See “Large I/O and performance” on page 225 for a list of query types.

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

This estimate is especially important if you are configuring a 16K pool for the first time on an active production server. Make the best possible estimate of simultaneous uses of the cache.

Some guidelines:

- If most I/O occurs in point queries using indexes to access a small number of rows, make the 16K pool relatively small, say about 10 to 20% of the cache size.
- If you estimate that a large percentage of the I/Os will use the 16K pool, configure 50 to 75% of the cache for 16K I/O.

Queries that use 16K I/O include any query that scans a table, uses the clustered index for range searches and order by, and queries that perform matching or nonmatching scans on covering nonclustered indexes.

- If you are not sure about the I/O size that will be used by your queries, configure about 20% of your cache space in a 16K pool, and use `showplan` and `statistics i/o` while you run your queries.

Examine the showplan output for the “Using 16K I/O” message. Check `statistics i/o` output to see how much I/O is performed.

- If you think that your typical application mix uses both 16K I/O and 2K I/O simultaneously, configure 30 to 40% of your cache space for 16K I/O.

Your optimum may be higher or lower, depending on the actual mix and the I/O sizes chosen by the query.

If many tables are accessed by both 2K I/O and 16K I/O, Adaptive Server cannot use 16K I/O, if any page from the extent is in the 2K cache. It performs 2K I/O on the other pages in the extent that are needed by the query. This adds to the I/O in the 2K cache.

After configuring for 16K I/O, check cache usage and monitor the I/O for the affected devices, using `sp_sysmon` or Adaptive Server Monitor. Also, use `showplan` and `statistics io` to observe your queries.

- Look for nested-loop join queries where an inner table would use 16K I/O, and the table is repeatedly scanned using the fetch-and-discard (MRU) strategy.

This can occur when neither table fits completely in cache. If increasing the size of the 16K pool allows the inner table to fit completely in cache, I/O can be significantly reduced. You might also consider binding the two tables to separate caches.

- Look for excessive 16K I/O, when compared to table size in pages.

For example, if you have an 8000-page table, and a 16K I/O table scan performs significantly more than 1000 I/Os to read this table, you may see improvement by re-creating the clustered index on this table.

- Look for times when large I/O is denied. Many times, this is because pages are already in the 2K pool, so the 2K pool will be used for the rest of the I/O for the query.

For a complete list of the reasons that large I/O cannot be used, see “When prefetch specification is not followed” on page 44 in the book *Performance and Tuning: Optimizer*.

Configuring buffer wash size

You can configure the wash area for each pool in each cache. If you set the wash size is set too high, Adaptive Server may perform unnecessary writes. If you set the wash area too small, Adaptive Server may not be able to find a clean buffer at the end of the buffer chain and may have to wait for I/O to complete before it can proceed. Generally, wash size defaults are correct and need to be adjusted only in large pools that have very high rates of data modification.

Adaptive Server allocates buffer pools in units of logical pages. For example, on a server using 2K logical pages, 8MB are allocated to the default data cache. By default this constitutes approximately 4096 buffers.

If you allocated the same 8MB for the default data cache on a server using a 16K logical page size, the default data cache is approximately 512 buffers. On a busy system, this small number of buffers might result in a buffer always being in the wash region, causing a slowdown for tasks requesting clean buffers.

In general, to obtain the same buffer management characteristics on larger page sizes as with 2K logical page sizes, you should scale the size of the caches to the larger page size. In other words, if you increase your logical page size by four times, your cache and pool sizes should be about four times larger as well.

Queries performing large I/O, extent-based reads and writes, and so on, benefit from the use of larger logical page sizes. However, as catalogs continue to be page-locked, there is greater contention and blocking at the page level on system catalogs.

Row and column copying for DOL tables will result in a greater slowdown when used for wide columns. Memory allocation to support wide rows and wide columns will marginally slow the server.

See the *System Administration Guide* for more information.

Overhead of pool configuration and binding objects

Configuring memory pools and binding objects to caches can affect users on a production system, so these activities are best performed during off-hours.

Pool configuration overhead

When a pool is created, deleted, or changed, the plans of all stored procedures and triggers that use objects bound to the cache are recompiled the next time they are run. If a database is bound to the cache, this affects all of the objects in a database.

There is a slight amount of overhead involved in moving buffers between pools.

Cache binding overhead

When you bind or unbind an object, all the object's pages that are currently in the cache are flushed to disk (if dirty) or dropped from the cache (if clean) during the binding process.

The next time the pages are needed by user queries, they must be read from the disk again, slowing the performance of the queries.

Adaptive Server acquires an exclusive lock on the table or index while the cache is being cleared, so binding can slow access to the object by other users. The binding process may have to wait until transactions complete to acquire the lock.

Note The fact that binding and unbinding objects from caches removes them from memory can be useful when tuning queries during development and testing.

If you need to check physical I/O for a particular table, and earlier tuning efforts have brought pages into cache, you can unbind and rebind the object. The next time the table is accessed, all pages used by the query must be read into the cache.

The plans of all stored procedures and triggers using the bound objects are recompiled the next time they are run. If a database is bound to the cache, this affects all the objects in the database.

Maintaining data cache performance for large I/O

When heap tables, clustered indexes, or nonclustered indexes have just been created, they show optimal performance when large I/O is being used. Over time, the effects of deletes, page splits, and page deallocation and reallocation can increase the cost of I/O. `optdiag` reports a statistic called “Large I/O efficiency” for tables and indexes.

When this value is 1, or close to 1, large I/O is very efficient. As the value drops, more I/O is required to access data pages needed for a query, and large I/O may be bringing pages into cache that are not needed by the query.

You need to consider rebuilding indexes when large I/O efficiency drops or activity in the pool increases due to increased 16K I/O.

When large I/O efficiency drops, you can:

- Run `reorg rebuild` on tables that use data-only-locking. You can also use `reorg rebuild` on the index of data-only-locked tables.
- For allpages-locked tables, drop and re-create the indexes.

For more information, see “Running `reorg` on tables and indexes” on page 343.

Diagnosing excessive I/O Counts

There are several reasons why a query that performs large I/O might require more reads than you anticipate:

- The cache used by the query has a 2K cache and other processes have brought pages from the table into the 2K cache.

If Adaptive Server finds that one of the pages it would read using 16K I/Os already in the 2K cache, it performs 2K I/O on the other pages in the extent that are required by the query.

- The first extent on each allocation unit stores the allocation page, so if a query needs to access all the pages on the extent, it must perform 2K I/O on the 7 pages that share the extent with the allocation page.

The other 31 extents can be read using 16K I/O. So, the minimum number of reads for an entire allocation unit is always 38, not 32.

- In nonclustered indexes and clustered indexes on data-only-locked tables, an extent may store both leaf-level pages and pages from higher levels of the index. 2K I/O is performed on the higher levels of indexes, and for leaf-level pages when few pages are needed by a query.

When a covering leaf-level scan performs 16K I/O, it is likely that some of the pages from some extents will be in the 2K cache. The rest of the pages in the extent will be read using 2K I/O.

Using `sp_sysmon` to check large I/O performance

The `sp_sysmon` output for each data cache includes information that can help you determine the effectiveness for large I/Os in the *Performance and Tuning: Monitoring and Analyzing for Performance* book:

- “Large I/O usage” on page 85 reports the number of large I/Os performed and denied and provides summary statistics.
- “Large I/O detail” on page 95 reports the total number of pages that were read into the cache by a large I/O and the number of pages that were actually accessed while they were in the cache.

Speed of recovery

As users modify data in Adaptive Server, only the transaction log is written to disk immediately, to ensure that given data or transactions can be recovered. The changed or “dirty” data and index pages stay in the data cache until one of these events causes them to be written to disk:

- The checkpoint process wakes up, determines that the changed data and index pages for a particular database need to be written to disk, and writes out all the dirty pages in each cache used by the database.

The combination of the setting for recovery interval and the rate of data modifications on your server determine how often the checkpoint process writes changed pages to disk.

- As pages move into the buffer wash area of the cache, dirty pages are automatically written to disk.
- Adaptive Server has spare CPU cycles and disk I/O capacity between user transactions, and the housekeeper wash task uses this time to write dirty buffers to disk.
- Recovery happens only on the default data cache.
- A user issues a checkpoint command.

You can use the checkpoint to identify one or more databases or use an all clause.

```
checkpoint [all | [dbname[, dbname[, dbname.....]]]
```

The combination of checkpoints, the housekeeper, and writes started at the wash marker has these benefits:

- Many transactions may change a page in the cache or read the page in the cache, but only one physical write is performed.
- Adaptive Server performs many physical writes at times when the I/O does not cause contention with user processes.

Tuning the recovery interval

The default recovery interval in Adaptive Server is five minutes per database. Changing the recovery interval can affect performance because it can impact the number of times Adaptive Server writes pages to disk.

Table 10-3 shows the effects of changing the recovery interval from its current setting on your system.

Table 10-3: Effects of recovery interval on performance and recovery time

Setting	Effects on performance	Effects on recovery
Lower	May cause more reads and writes and may lower throughput. Adaptive Server will write dirty pages to the disk more often. Any checkpoint I/O “spikes” will be smaller.	Recovery period will be very short.
Higher	Minimizes writes and improves system throughput. Checkpoint I/O spikes will be higher.	Automatic recovery may take more time on start-up. Adaptive Server may have to reapply a large number of transaction log records to the data pages.

See the *System Administration Guide* for information on setting the recovery interval. `sp_sysmon` reports the number and duration of checkpoints.

See “Recovery management” on page 99 in the book *Performance and Tuning: Monitoring and Analyzing for Performance*.

Effects of the housekeeper wash task on recovery time

Adaptive Server’s housekeeper wash task automatically begins cleaning dirty buffers during the server’s idle cycles. If the task is able to flush all active buffer pools in all configured caches, it wakes up the checkpoint process. This may result in faster checkpoints and shorter database recovery time.

System Administrators can use the housekeeper free write percent configuration parameter to tune or disable the housekeeper wash task. This parameter specifies the maximum percentage by which the housekeeper task can increase database writes.

For more information on tuning the housekeeper and the recovery interval, see “Recovery management” on page 99 in the book *Performance and Tuning: Monitoring and Analyzing for Performance*.

Auditing and performance

Heavy auditing can affect performance as follows:

- Audit records are written first to a queue in memory and then to the subsecurity database. If the database shares a disk used by other busy databases, it can slow performance.
- If the in-memory audit queue fills up, the user processes that generate audit records sleep. See Figure 10-5 on page 245.

Sizing the audit queue

The size of the audit queue can be set by a System Security Officer. The default configuration is as follows:

- A single audit record requires a minimum of 32 bytes, up to a maximum of 424 bytes.

This means that a single data page stores between 4 and 80 records.

- The default size of the audit queue is 100 records, requiring approximately 42K.

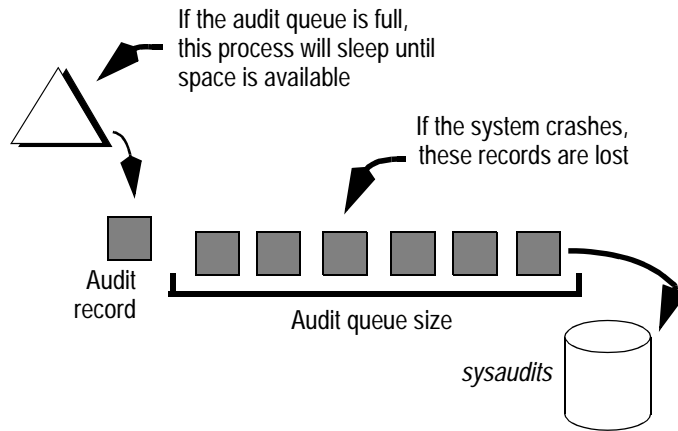
The minimum size of the queue is 1 record; the maximum size is 65,335 records.

There are trade-offs in sizing the audit queue, as shown in Figure 10-5.

If the audit queue is large, so that you do not risk having user processes sleep, you run the risk of losing any audit records in memory if there is a system failure. The maximum number of records that can be lost is the maximum number of records that can be stored in the audit queue.

If security is your chief concern, keep the queue small. If you can risk the loss of more audit records, and you require high performance, make the queue larger.

Increasing the size of the in-memory audit queue takes memory from the total memory allocated to the data cache.

Figure 10-5: Trade-offs in auditing and performance

Auditing performance guidelines

- Heavy auditing slows overall system performance. Audit only the events you need to track.
- If possible, place the sysaudits database on its own device. If that is not possible, place it on a device that is not used for your most critical applications.

Text and images pages

Text and image pages can use large portions of memory and are commonly known as space wastage. They exist as long as a parent data row points to the text and image pages. These pages come into existence when a null update is done against the columns.

Find the current status for the table:

```
sp_help<table name>
```

The text and image pages can be deallocated to open the space they occupy. Use the `sp_chgattribute`:

```
sp_chgattribute <table name>, "deallocate_first_txtpg",1
```

This switches the deallocation on. To switch the deallocation off:

```
sp_chgattribute <table name>, "deallocate_first_txtpg",0
```

Determining Sizes of Tables and Indexes

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

It contains the following sections:

Topic	Page
Why object sizes are important to query tuning	247
Tools for determining the sizes of tables and indexes	248
Effects of data modifications on object sizes	249
Using optdiag to display object sizes	249
Using sp_spaceused to display object size	250
Using sp_estspace to estimate object size	252
Using formulas to estimate object size	254

Why object sizes are important to query tuning

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 3, “Using Statistics to Improve Performance,” in the book *Performance and Tuning: Monitoring and Analyzing for Performance* describes how to use statistics io to examine the I/O performed.
- Understand the optimizer’s choice of query plan. Adaptive Server’s 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, you can 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. You can improve performance by distributing database objects across physical devices so that reads and writes to disk are evenly distributed. Object placement is described in Chapter 6, “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 percent 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.
- Do capacity planning. Whether you are designing a new system or planning for growth of an existing system, you need to know the space requirements in order to plan for physical disks and memory needs.
- Understand output from Adaptive Server Monitor and from `sp_sysmon` reports on physical I/O.

Tools for determining the sizes of tables and indexes

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

- The utility program `optdiag` displays the sizes and many other statistics for tables and indexes. For information on using `optdiag`, see Chapter 6, “Statistics Tables and Displaying Statistics with `optdiag`.” in the book *Performance and Tuning: Monitoring and Analyzing for Performance*.
- The system procedure `sp_spaceused` reports on the current size of an existing table and any indexes.
- The system procedure `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. The `sp_spaceused` and `optdiag` commands report actual space usage. The other methods presented in this chapter provide size estimates. For partitioned tables, the system procedure `sp_helppartition` reports on the number of pages stored on each partition of the table. See “Getting information about partitions” on page 111 for information.

Effects of data modifications on object sizes

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

- When you insert a row that needs 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 engaged in query tuning, *optdiag* provides the best tool for viewing all the statistics that 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 6, “Statistics Tables and Displaying Statistics with *optdiag*,” in the book *Performance and Tuning: Monitoring and Analyzing for Performance* for more information.

Advantages of *optdiag*

The advantages of *optdiag* are:

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

Disadvantages of *optdiag*

The disadvantages of *optdiag* are:

- It produces a lot of output, so if you need only a single piece of information, such as the number of pages in the table, other methods are faster and have 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 11-1 explains the headings in *sp_spaceused* output.

Table 11-1: *sp_spaceused* output

Column	Meaning
<i>rowtotal</i>	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 <code>select count(*)</code> .
<i>reserved</i>	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 <code>data</code> , <code>index_size</code> , and <code>unused</code> .
<i>data</i>	Reports the kilobytes on pages used by the table.
<i>index_size</i>	Reports the total kilobytes on pages used by the indexes.
<i>unused</i>	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:

- 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.

Disadvantages of `sp_spaceused`

The disadvantages of `sp_spaceused` are:

- It may report inaccurate counts for row total and space usage.
- Output is in kilobytes, while most query-tuning activities use pages as a unit of measure.

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 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 in the *Adaptive Server Reference Manual*.

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 using `sp_estspace` to estimate the sizes of objects are:

- `sp_estspace` provides a quick, easy way to perform initial capacity planning and to plan for table and index growth.
- `sp_estspace` helps you estimate the number of index levels.
- `sp_estspace` can be used to estimate future disk space, cache space, and memory requirements.

Disadvantages of `sp_estspace`

The disadvantages of using `sp_estspace` to estimate the sizes of objects are:

- 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

Use the formulas in this section to 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 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, 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 268, and “max_rows_per_page” on page 269.

The formulas in this section use the maximum size for variable-length character and binary data. To use the average size instead of the maximum size, see “Using average sizes for variable fields” on page 269.

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 270 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 non-leaf 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. See in the *System Administration Guide*.

Storage sizes for datatypes

The storage sizes for datatypes are shown in Table 11-2:

Table 11-2: 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. If you want to use average values instead, see “Using average sizes for variable fields” on page 269.

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 257 for tables that use allpages-locking.
- See “Calculating the sizes of data-only-locked tables” on page 263 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 divided into two sets 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.

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. Once you compute the number of data pages in step 2, go to step 6 to add the number of OAM pages.

Step 1: 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

Use this formula if the table contains only fixed-length columns, and all are defined as NOT NULL.

Formula

$$\begin{array}{r}
 4 \quad (\text{Overhead}) \\
 + \quad \text{Sum of bytes in all fixed-length columns} \\
 \hline
 = \text{Data row size}
 \end{array}$$

Some variable-length columns

Use this formula if the table contains any variable-length columns or columns that allow null values.

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
<hr/> = Subtotal	<hr/> 154
+ (Subtotal / 256) + 1 (Overhead)	1
+ Number of variable-length columns + 1	3
+ 2 (Overhead)	2
<hr/> = Data row size	<hr/> 160

Step 2: Compute the number of data pages

Formula

$2016 / \text{Data row size} = \text{Number of data rows per page}$

$\text{Number of rows} / \text{Rows per page} = \text{Number of data pages required}$

Example

$$\begin{aligned}
 2016 / 160 &= 12 \text{ data rows per page} \\
 9,000,000 / 12 &= 750,000 \text{ data pages}
 \end{aligned}$$

Step 3: 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
<hr style="width: 100px; margin-left: 0;"/> = Clustered row size	<hr style="width: 100px; margin-left: 0;"/> 9

Some variable-length columns

$$\begin{aligned}
 &5 \text{ (Overhead)} \\
 + &\text{ Sum of bytes in the fixed-length index keys} \\
 + &\text{ Sum of bytes in variable-length index keys} \\
 \hline
 &= \text{Subtotal} \\
 \\
 + &\text{ (Subtotal / 256) + 1 (Overhead)} \\
 + &\text{ Number of variable-length columns + 1} \\
 + &2 \text{ (Overhead)} \\
 \hline
 &= \text{Clustered index row size}
 \end{aligned}$$

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

Step 4: 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

Formula

$$\text{No. of rows / No. of CI rows per page} = \text{No. of index pages at next level} \quad 750,000 / 222 = 3379$$

Example

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

$$\text{No. of index pages at last level} / \text{No. of clustered index rows per page} = \text{No. of index pages at next level}$$

Example

$$3379 / 222 = 16 \text{ index pages (Level 1)}$$

$$16 / 222 = 1 \text{ index page (Level 2)}$$

Step 5: 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

Step 6: 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

$$\text{Number of reserved data pages} / 63,750 = \text{Minimum OAM pages} \quad 750,000 / 63,750 = 12$$

$$\text{Number of reserved data pages} / 2000 = \text{Maximum OAM pages} \quad 750,000 / 2000 = 376$$

Example

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

Step 8: Calculate the number of leaf pages in the index

Formula		Example
$(2016 / \text{leaf row size})$	$=$	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
		$9,000,000 / 56 = 160,715$

Step 9: Calculate the size of the non-leaf rows

Formula	Example
Size of leaf index row	36
+ 4 Overhead	+ 4
<hr style="width: 50px; margin-left: 0;"/>	<hr style="width: 50px; margin-left: 0;"/>
= Size of non-leaf row	40

Step 10: Calculate the number of non-leaf pages

Formula	Example
$(2016 / \text{Size of non-leaf row}) - 2$	$=$
	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)

Step 11: 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

Index Levels	Pages	Pages	Rows
3	+	+	2
2	+	+	70
1	+	+	3349
0	+	+	160715
		<hr/>	
Total number of 2K data pages used		164137	

Step 12: 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			<hr/>	<hr/>
			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 columns sizes and index as the previous example. See “Tables and indexes used in the formulas” on page 257 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.

Step 1: 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

Use this formula if the table contains only fixed-length columns defined as NOT NULL:

$$\begin{array}{r}
 6 \quad (\text{Overhead}) \\
 + \quad \text{Sum of bytes in all fixed-length columns} \\
 \hline
 \text{Data row size}
 \end{array}$$

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

Use this formula if the table contains variable-length columns or columns that allow null values:

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

Step 2: Compute the number of data pages

Formula

$2002 / \text{Data row size} = \text{Number of data rows per page}$

$\text{Number of rows} / \text{Rows per page} = \text{Number of data pages required}$

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

Example

$$2002 / 162 = 12 \text{ data rows per page}$$

$$9,000,000 / 12 = 750,000 \text{ data pages}$$

Step 3: 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

Step 4: 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 Use this formula if the index contains only fixed-length keys defined as NOT NULL:

$$9 \text{ (Overhead)}$$

$$\frac{+ \quad \text{Sum of fixed-length keys}}{\text{Size of index row}}$$

Some variable-length keys

Use this formula if the index contains any variable-length keys or columns that allow null values:

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

Step 5: Calculate the number of leaf pages in the index

Formula

$$\begin{aligned} 2002 / \text{Size of index row} &= \text{No. of rows per page} \\ \text{No. of rows in table} / \text{No. of rows per page} &= \text{No. of leaf pages} \end{aligned}$$

Example

$$\begin{aligned} 2002 / 35 &= 57 \text{ Nonclustered index rows per page} \\ 9,000,000 / 57 &= 157,895 \text{ leaf pages} \end{aligned}$$

Step 6: Calculate the number of non-leaf pages in the index

Formula

$$\text{No. of leaf pages} / \text{No. of index rows per page} = \text{No. of pages at next level}$$

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

$$\text{No. of index pages at previous level} / \text{No. of non-leaf index rows per page} = \text{No. of index pages at next level}$$

Example

$$157895/57 = 2771 \quad \text{Index pages, level 1}$$

Example

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

Step 7: 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	

Step 8: 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
Nonclustered index pages	160716	160716
OAM pages	+	+
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:

- The 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...lock` 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.

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 “Step 2: Compute the number of data pages” on page 258 and “Step 8: Calculate the number of leaf pages in the index” on page 262.

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.

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. For more information, see “Leaving space for forwarded rows and inserts” on page 194.

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 “Step 2: Compute the number of data pages” on page 258 and “Step 8: Calculate the number of leaf pages in the index” on page 262.

Using average sizes for variable fields

All of the formulas use the maximum size of the variable-length fields.

`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.

Very small rows

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. Of course, 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:

$\text{Data length} / 1800 = \text{Number of 2K pages}$

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 the information in Table 11-3 to estimate the additional pages required to store pointer information for data in LOB columns.

Table 11-3: 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.

How Indexes Work

This chapter describes how Adaptive Server stores indexes and how it uses indexes to speed data retrieval for select, update, delete, and insert operations.

Topic	Page
Types of indexes	274
Clustered indexes on allpages-locked tables	276
Nonclustered indexes	285
Index covering	291
Indexes and caching	295

Indexes are the most important physical design element in improving database performance:

- Indexes help prevent table scans. Instead of reading hundreds of data pages, a few index pages and data pages can satisfy many queries.
- For some queries, data can be retrieved from a nonclustered index without ever accessing the data rows.
- Clustered indexes can randomize data inserts, avoiding insert “hot spots” on the last page of a table.
- Indexes can help avoid sorts, if the index order matches the order of columns in an order by clause.

In addition to their performance benefits, indexes can enforce the uniqueness of data.

Indexes are database objects that can be created for a table to speed direct access to specific data rows. Indexes store the values of the key(s) that were named when the index was created, and logical pointers to the data pages or to other index pages.

Although indexes speed data retrieval, they can slow down data modifications, since most changes to the data also require updating the indexes. Optimal indexing demands:

- An understanding of the behavior of queries that access unindexed heap tables, tables with clustered indexes, and tables with nonclustered indexes
- An understanding of the mix of queries that run on your server
- An understanding of the Adaptive Server optimizer

Types of indexes

Adaptive Server provides two types of indexes:

- Clustered indexes, where the table data is physically stored in the order of the keys on the index:
 - For allpages-locked tables, rows are stored in key order on pages, and pages are linked in key order.
 - For data-only-locked tables, indexes are used to direct the storage of data on rows and pages, but strict key ordering is not maintained.
- Nonclustered indexes, where the storage order of data in the table is not related to index keys

You can create only one clustered index on a table because there is only one possible physical ordering of the data rows. You can create up to 249 nonclustered indexes per table.

A table that has no clustered index is called a heap. The rows in the table are in no particular order, and all new rows are added to the end of the table. Chapter 8, “Data Storage,” discusses heaps and SQL operations on heaps.

Index pages

Index entries are stored as rows on index pages in a format similar to the format used for data rows on data pages. Index entries store the key values and pointers to lower levels of the index, to the data pages, or to individual data rows.

Adaptive Server uses B-tree indexing, so each node in the index structure can have multiple children.

Index entries are usually much smaller than a data row in a data page, and index pages are much more densely populated than data pages. If a data row has 200 bytes (including row overhead), there are 10 rows per page.

An index on a 15-byte field has about 100 rows per index page (the pointers require 4–9 bytes per row, depending on the type of index and the index level).

Indexes can have multiple levels:

- Root level
- Leaf level
- Intermediate level

Root level

The root level is the highest level of the index. There is only one root page. If an allpages-locked table is very small, so that the entire index fits on a single page, there are no intermediate or leaf levels, and the root page stores pointers to the data pages.

Data-only-locked tables always have a leaf level between the root page and the data pages.

For larger tables, the root page stores pointers to the intermediate level index pages or to leaf-level pages.

Leaf level

The lowest level of the index is the leaf level. At the leaf level, the index contains a key value for each row in the table, and the rows are stored in sorted order by the index key:

- For clustered indexes on allpages-locked tables, the leaf level is the data. No other level of the index contains one index row for each data row.
- For nonclustered indexes and clustered indexes on data-only-locked tables, the leaf level contains the index key value for each row, a pointer to the page where the row is stored, and a pointer to the rows on the data page.

The leaf level is the level just above the data; it contains one index row for each data row. Index rows on the index page are stored in key value order.

Intermediate level

All levels between the root and leaf levels are intermediate levels. An index on a large table or an index using long keys may have many intermediate levels. A very small allpages-locked table may not have an intermediate level at all; the root pages point directly to the leaf level.

Index Size

Table 12-1 describes the new limits for index size for APL and DOL tables:

Table 12-1: Index row-size limit

Page size	User-visible index row-size limit	Internal index row-size limit
2K (2048 bytes)	600	650
4K (4096bytes)	1250	1310
8K (8192 bytes)	2600	2670
16K (16384 bytes)	5300	5390

Because you can create tables with columns wider than the limit for the index key, these columns become non-indexable. For example, if you perform the following on a 2K page server, then try to create an index on c3, the command fails and Adaptive Server issues an error message because column c3 is larger than the index row-size limit (600 bytes).

```
create table t1 (
  c1 int
  c2 int
  c3 char(700))
```

“Non-indexable” does not mean that you cannot use these columns in search clauses. Even though a column is non-indexable (as in c3, above), you can still create statistics for it. Also, if you include the column in a where clause, it will be evaluated during optimization.

Clustered indexes on allpages-locked tables

In clustered indexes on allpages-locked tables, leaf-level pages are also the data pages, and all rows are kept in physical order by the keys.

Physical ordering means that:

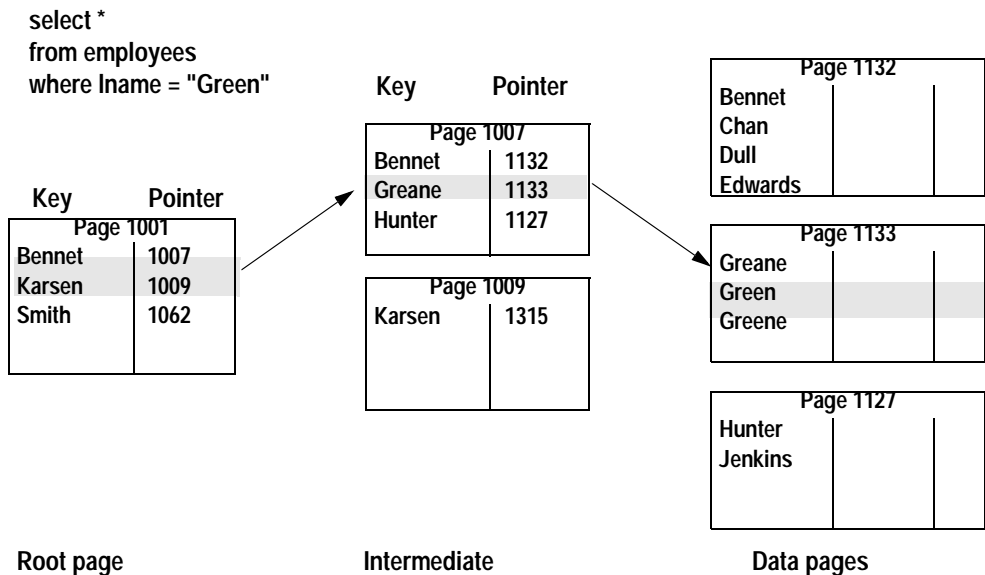
- All entries on a data page are in index key order.
- By following the “next page” pointers on the data pages, Adaptive Server reads the entire table in index key order.

On the root and intermediate pages, each entry points to a page on the next level.

Clustered indexes and select operations

To select a particular last name using a clustered index, Adaptive Server first uses sysindexes to find the root page. It examines the values on the root page and then follows page pointers, performing a binary search on each page it accesses as it traverses the index. See Figure 12-1 below.

Figure 12-1: Selecting a row using a clustered index, allpages-locked table



On the root level page, “Green” is greater than “Bennet,” but less than Karsen, so the pointer for “Bennet” is followed to page 1007. On page 1007, “Green” is greater than “Greane,” but less than “Hunter,” so the pointer to page 1133 is followed to the data page, where the row is located and returned to the user.

This retrieval via the clustered index requires:

- One read for the root level of the index
- One read for the intermediate level
- One read for the data page

These reads may come either from cache (called a **logical read**) or from disk (called a **physical read**). On tables that are frequently used, the higher levels of the indexes are often found in cache, with lower levels and data pages being read from disk.

Clustered indexes and insert operations

When you insert a row into an allpages-locked table with a clustered index, the data row must be placed in physical order according to the key value on the table.

Other rows on the data page move down on the page, as needed, to make room for the new value. As long as there is room for the new row on the page, the insert does not affect any other pages in the database.

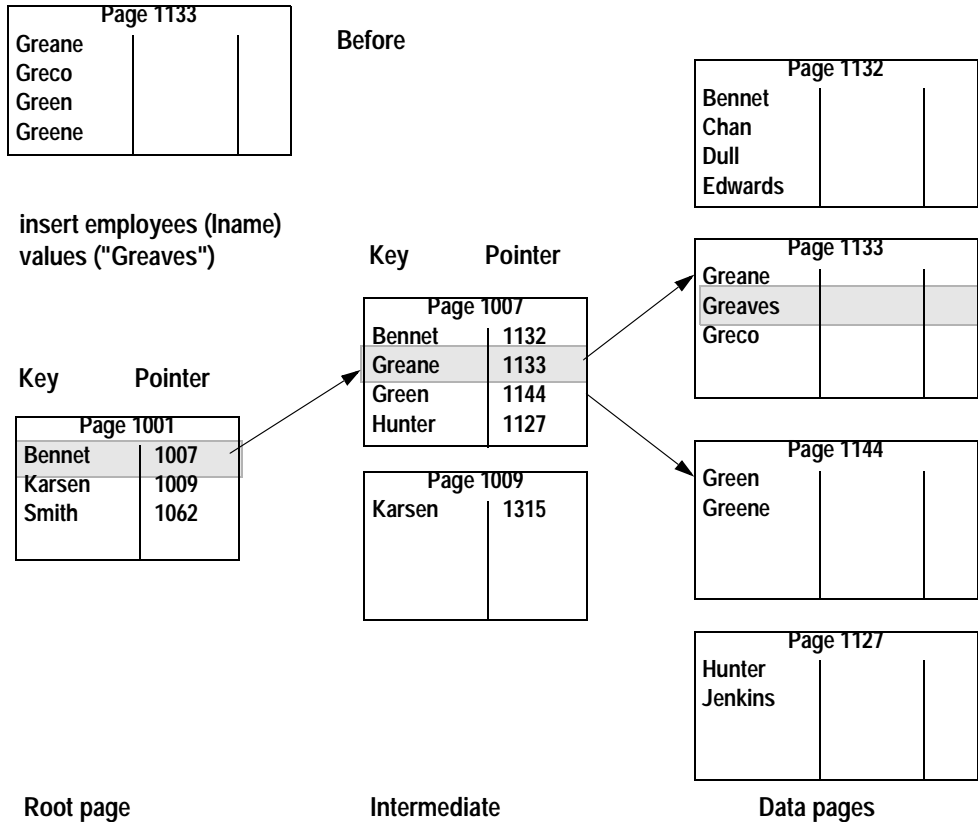
The clustered index is used to find the location for the new row.

Figure 12-2 shows a simple case where there is room on an existing data page for the new row. In this case, the key values in the index do not need to change.

See “Exceptions to page splitting” on page 280.

In Figure 12-3, the page split requires adding a new row to an existing index page, page 1007.

Figure 12-3: Page splitting in an allpages-locked table with a clustered index



Exceptions to page splitting

There are exceptions to 50-50 page splits:

- If you insert a huge row that cannot fit on either the page before or the page after the page that requires splitting, two new pages are allocated, one for the huge row and one for the rows that follow it.

- If possible, Adaptive Server keeps duplicate values together when it splits pages.
- If Adaptive Server detects that all inserts are taking place at the end of the page, due to a increasing key value, the page is not split when it is time to insert a new row that does not fit at the bottom of the page. Instead, a new page is allocated, and the row is placed on the new page.
- If Adaptive Server detects that inserts are taking place in order at other locations on the page, the page is split at the insertion point.

Page splitting on index pages

If a new row needs to be added to a full index page, the page split process on the index page is similar to the data page split.

A new page is allocated, and half of the index rows are moved to the new page.

A new row is inserted at the next highest level of the index to point to the new index page.

Performance impacts of page splitting

Page splits are expensive operations. In addition to the actual work of moving rows, allocating pages, and logging the operations, the cost is increased by:

- Updating the clustered index itself
- Updating the page pointers on adjacent pages to maintain page linkage
- Updating all nonclustered index entries that point to the rows affected by the split

When you create a clustered index for a table that will grow over time, you may want to use `fillfactor` to leave room on data pages and index pages. This reduces the number of page splits for a time.

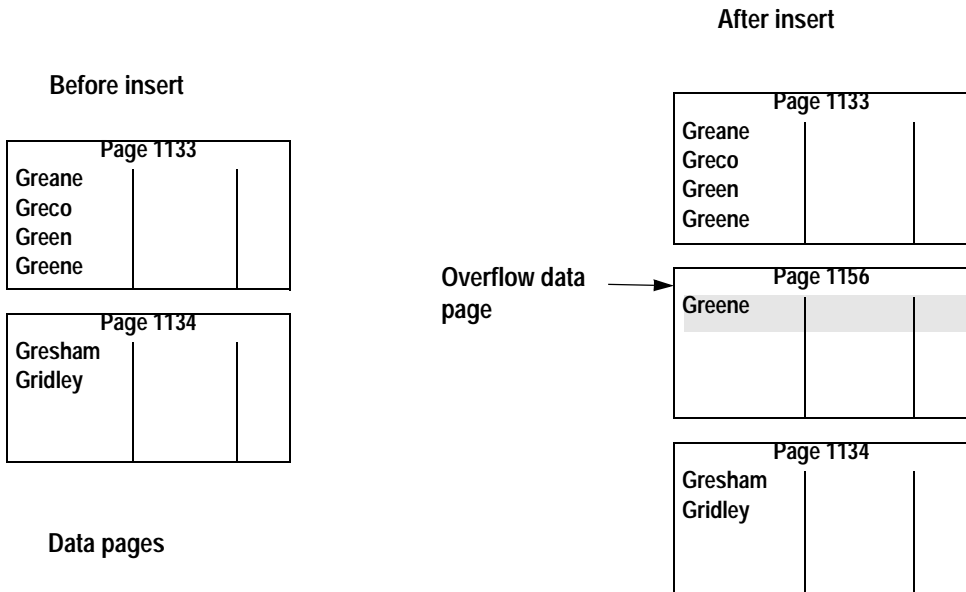
See “Choosing space management properties for indexes” on page 318.

Overflow pages

Special overflow pages are created for nonunique clustered indexes on allpages-locked tables when a newly inserted row has the same key as the last row on a full data page. A new data page is allocated and linked into the page chain, and the newly inserted row is placed on the new page (see Figure 12-4).

Figure 12-4: Adding an overflow page to a clustered index, allpages-locked table

```
insert employees (lname)
values('Greene')
```



The only rows that will be placed on this overflow page are additional rows with the same key value. In a nonunique clustered index with many duplicate key values, there can be numerous overflow pages for the same value.

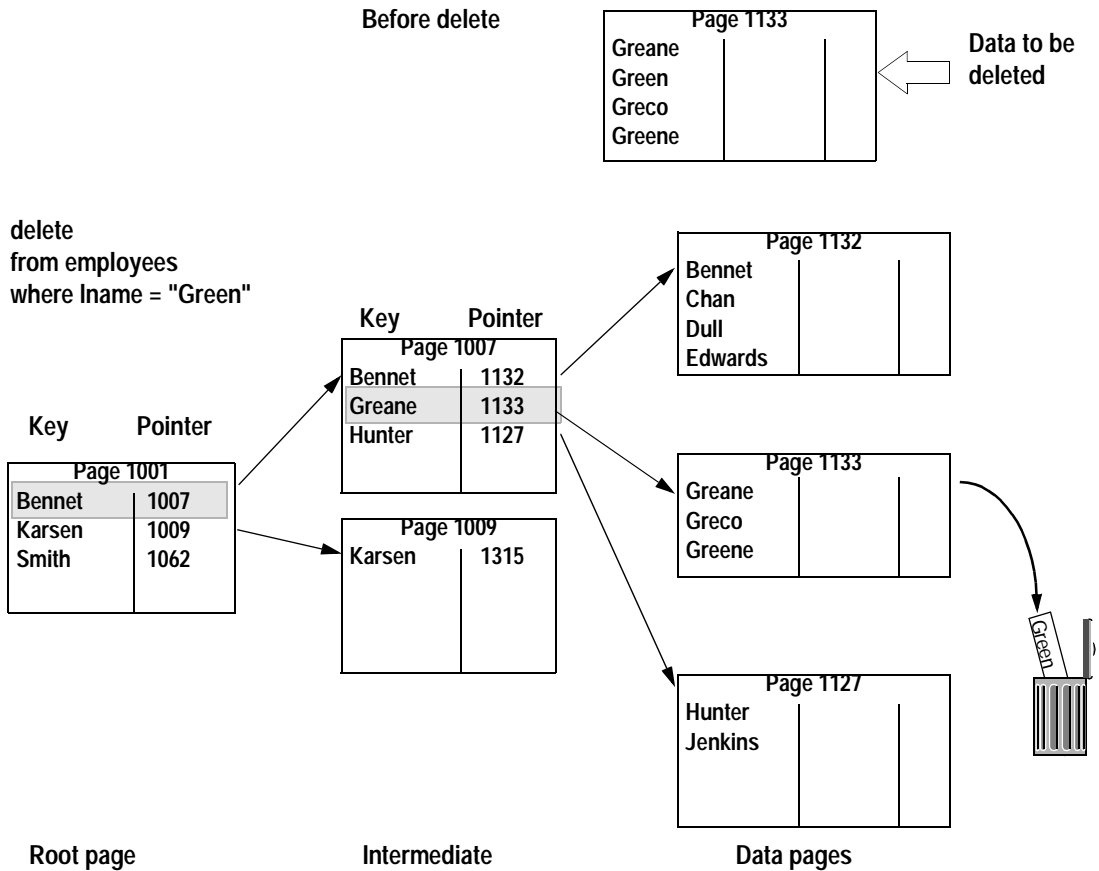
The clustered index does not contain pointers directly to overflow pages. Instead, the next page pointers are used to follow the chain of overflow pages until a value is found that does not match the search value.

Clustered indexes and delete operations

When you delete a row from an allpages-locked table that has a clustered index, other rows on the page move up to fill the empty space so that the data remains contiguous on the page.

Figure 12-5 shows a page that has four rows before a delete operation removes the second row on the page. The two rows that follow the deleted row are moved up.

Figure 12-5: Deleting a row from a table with a clustered index



Deleting the last row on a page

If you delete the last row on a data page, the page is deallocated and the next and previous page pointers on the adjacent pages are changed.

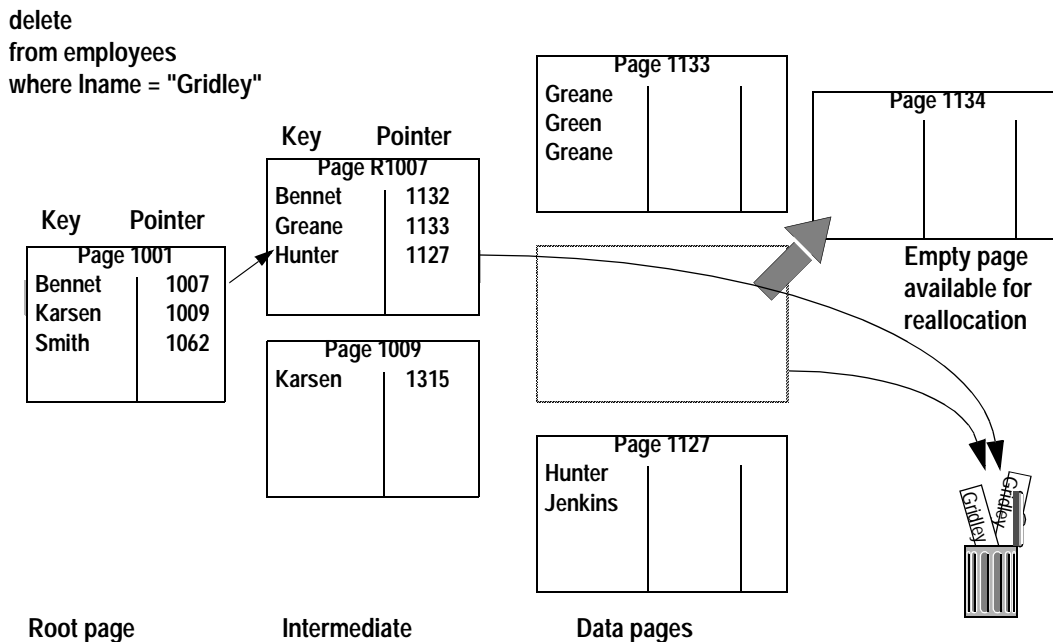
The rows that point to that page in the leaf and intermediate levels of the index are removed.

If the deallocated data page is on the same extent as other pages belonging to the table, it can be used again when that table needs an additional page.

If the deallocated data page is the last page on the extent that belongs to the table, the extent is also deallocated and becomes available for the expansion of other objects in the database.

In Figure 12-6, which shows the table after the deletion, the pointer to the deleted page has been removed from index page 1007 and the following index rows on the page have been moved up to keep the used space contiguous.

Figure 12-6: Deleting the last row on a page (after the delete)



Index page merges

If you delete a pointer from an index page, leaving only one row on that page, the row is moved onto an adjacent page, and the empty page is deallocated. The pointers on the parent page are updated to reflect the changes.

Nonclustered indexes

The B-tree works much the same for nonclustered indexes as it does for clustered indexes, but there are some differences. In nonclustered indexes:

- The leaf pages are not the same as the data pages.
- The leaf level stores one key-pointer pair for *each row* in the table.
- The leaf-level pages store the index keys and page pointers, plus a pointer to the row offset table on the data page. This combination of page pointer plus the row offset number is called the **row ID**.
- The root and intermediate levels store index keys and page pointers to other index pages. They also store the row ID of the key's data row.

With keys of the same size, nonclustered indexes require more space than clustered indexes.

Leaf pages revisited

The leaf page of an index is the lowest level of the index where all of the keys for the index appear in sorted order.

In clustered indexes on allpages-locked tables, the data rows are stored in order by the index keys, so by definition, the data level is the leaf level. There is no other level of the clustered index that contains one index row for each data row. Clustered indexes on allpages-locked tables are sparse indexes.

The level above the data contains one pointer for every data *page*, not data *row*.

In nonclustered indexes and clustered indexes on data-only-locked tables, the level just above the data is the leaf level: it contains a key-pointer pair for each data row. These indexes are dense. At the level above the data, they contain one index row for each data row.

Nonclustered index structure

The table in Figure 12-7 shows a nonclustered index on `lname`. The data rows at the far right show pages in ascending order by `employee_id` (10, 11, 12, and so on) because there is a clustered index on that column.

The root and intermediate pages store:

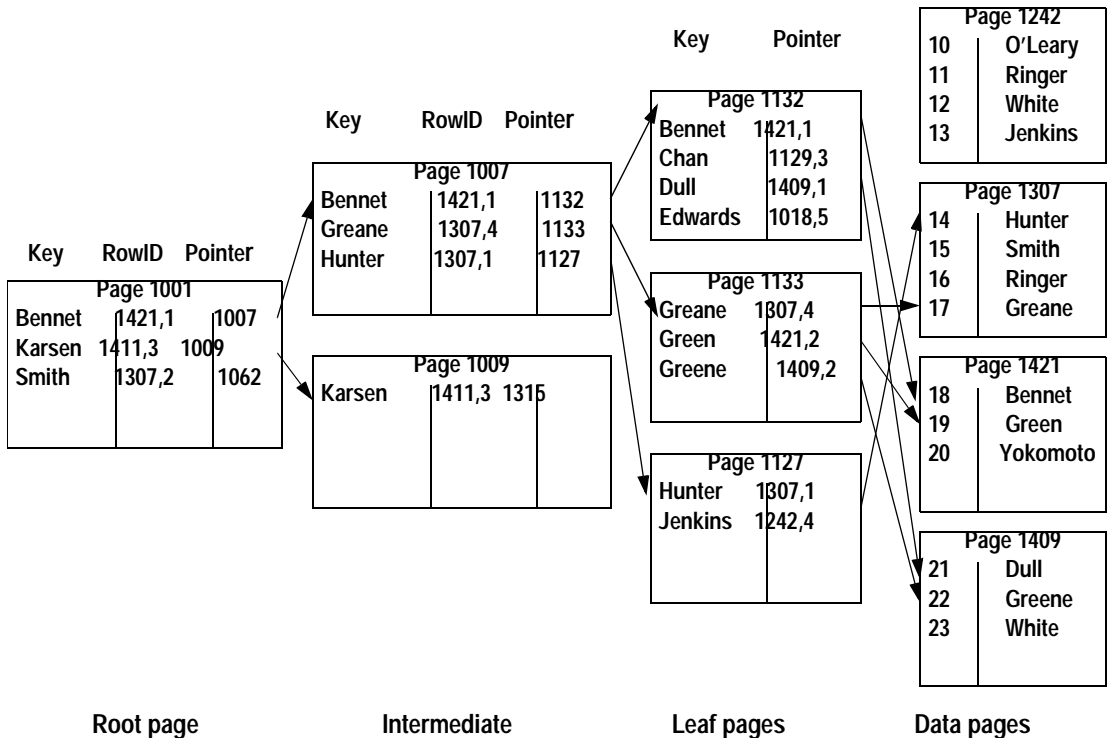
- The key value
- The row ID
- The pointer to the next level of the index

The leaf level stores:

- The key value
- The row ID

The row ID in higher levels of the index is used for indexes that allow duplicate keys. If a data modification changes the index key or deletes a row, the row ID positively identifies all occurrences of the key at all index levels.

Figure 12-7: Nonclustered index structure



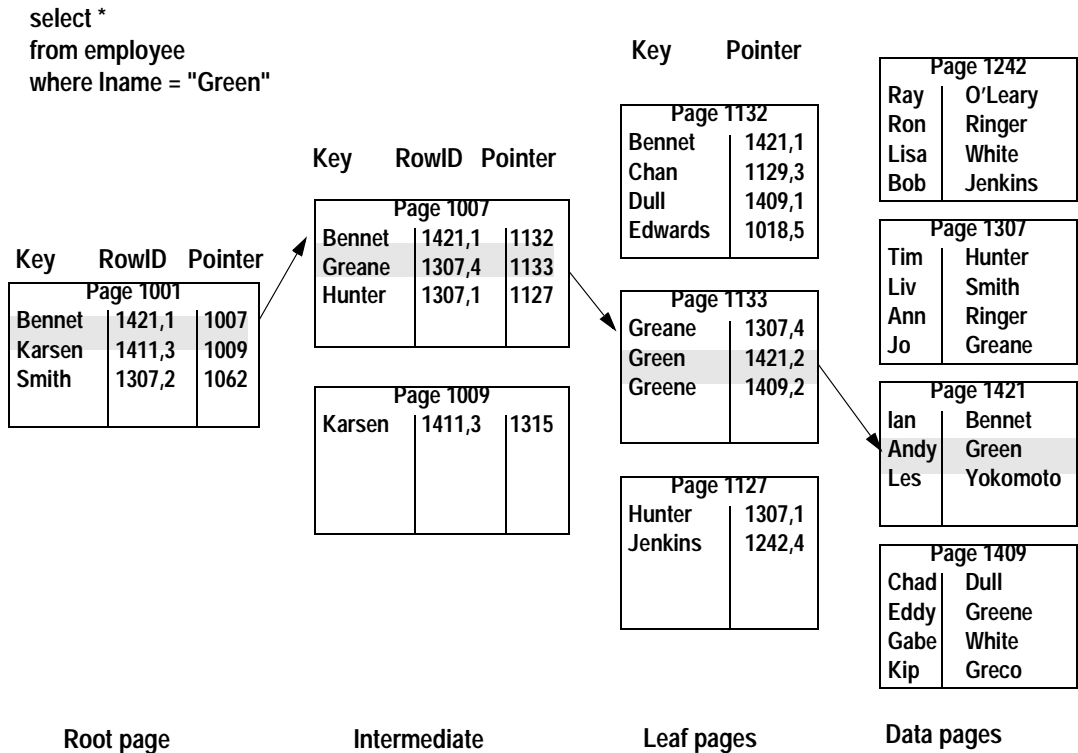
Nonclustered indexes and select operations

When you select a row using a nonclustered index, the search starts at the root level. `sysindexes.root` stores the page number for the root page of the nonclustered index.

In Figure 12-8, “Green” is greater than “Bennet,” but less than “Karsen,” so the pointer to page 1007 is followed.

“Green” is greater than “Greene,” but less than “Hunter,” so the pointer to page 1133 is followed. Page 1133 is the leaf page, showing that the row for “Green” is row 2 on page 1421. This page is fetched, the “2” byte in the offset table is checked, and the row is returned from the byte position on the data page.

Figure 12-8: Selecting rows using a nonclustered index



Nonclustered index performance

The query in Figure 12-8 requires the following I/O:

- One read for the root level page
- One read for the intermediate level page
- One read for the leaf-level page
- One read for the data page

If your applications use a particular nonclustered index frequently, the root and intermediate pages will probably be in cache, so only one or two physical disk I/Os need to be performed.

Nonclustered indexes and insert operations

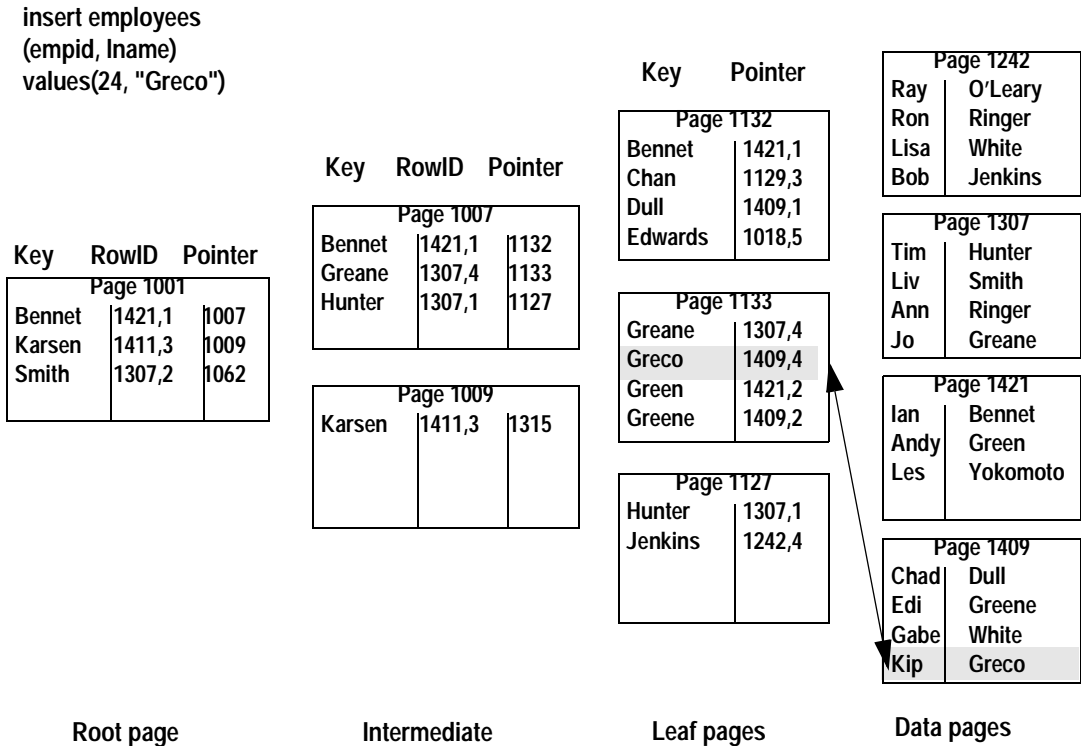
When you insert rows into a heap that has a nonclustered index and no clustered index, the insert goes to the last page of the table.

If the heap is partitioned, the insert goes to the last page on one of the partitions. Then, the nonclustered index is updated to include the new row.

If the table has a clustered index, it is used to find the location for the row. The clustered index is updated, if necessary, and each nonclustered index is updated to include the new row.

Figure 12-9 shows an insert into a heap table with a nonclustered index. The row is placed at the end of the table. A row containing the new key value and the row ID is also inserted into the leaf level of the nonclustered index.

Figure 12-9: An insert into a heap table with a nonclustered index

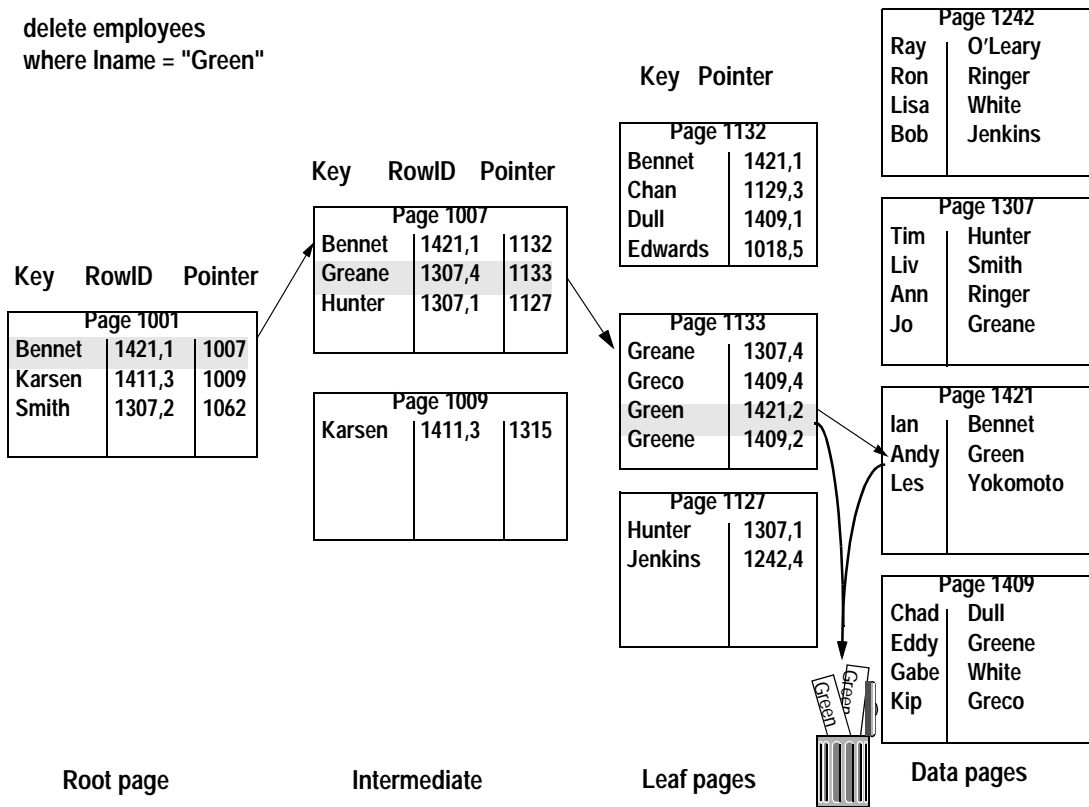


Nonclustered indexes and delete operations

When you delete a row from a table, the query can use a nonclustered index on the columns in the where clause to locate the data row to delete, as shown in Figure 12-10.

The row in the leaf level of the nonclustered index that points to the data row is also removed. If there are other nonclustered indexes on the table, the rows on the leaf level of those indexes are also deleted.

Figure 12-10: Deleting a row from a table with a nonclustered index



If the delete operation removes the last row on the data page, the page is deallocated and the adjacent page pointers are adjusted in all pages-locked tables. Any references to the page are also deleted in higher levels of the index.

If the delete operation leaves only a single row on an index intermediate page, index pages may be merged, as with clustered indexes.

See “Index page merges” on page 285.

There is no automatic page merging on data pages, so if your applications make many random deletes, you may end up with data pages that have only a single row, or a few rows, on a page.

Clustered indexes on data-only-locked tables

Clustered indexes on data-only-locked tables are structured like nonclustered indexes. They have a leaf level above the data pages. The leaf level contains the key values and row ID for each row in the table.

Unlike clustered indexes on allpages-locked tables, the data rows in a data-only-locked table are not necessarily maintained in exact order by the key. Instead, the index directs the placement of rows to pages that have adjacent or nearby keys.

When a row needs to be inserted in a data-only-locked table with a clustered index, the insert uses the clustered index key just before the value to be inserted. The index pointers are used to find that page, and the row is inserted on the page if there is room. If there is not room, the row is inserted on a page in the same allocation unit, or on another allocation unit already used by the table.

To provide nearby space for maintaining data clustering during inserts and updates to data-only-locked tables, you can set space management properties to provide space on pages (using `fillfactor` and `exp_row_size`) or on allocation units (using `reservepagegap`).

See Chapter 9, “Setting Space Management Properties.”

Index covering

Index covering can produce dramatic performance improvements when all columns needed by the query are included in the index.

You can create indexes on more than one key. These are called *composite indexes*. Composite indexes can have up to 31 columns adding up to a maximum 600 bytes.

If you create a composite nonclustered index on each column referenced in the query's select list and in any where, having, group by, and order by clauses, the query can be satisfied by accessing only the index.

Since the leaf level of a nonclustered index or a clustered index on a data-only-locked table contains the key values for each row in a table, queries that access only the key values can retrieve the information by using the leaf level of the nonclustered index as if it were the actual table data. This is called index covering.

There are two types of index scans that can use an index that covers the query:

- The matching index scan
- The nonmatching index scan

For both types of covered queries, the index keys must contain all the columns named in the query. Matching scans have additional requirements.

“Choosing composite indexes” on page 312 describes query types that make good use of covering indexes.

Covering matching index scans

Lets you skip the last read for each row returned by the query, the read that fetches the data page.

For point queries that return only a single row, the performance gain is slight — just one page.

For range queries, the performance gain is larger, since the covering index saves one read for each row returned by the query.

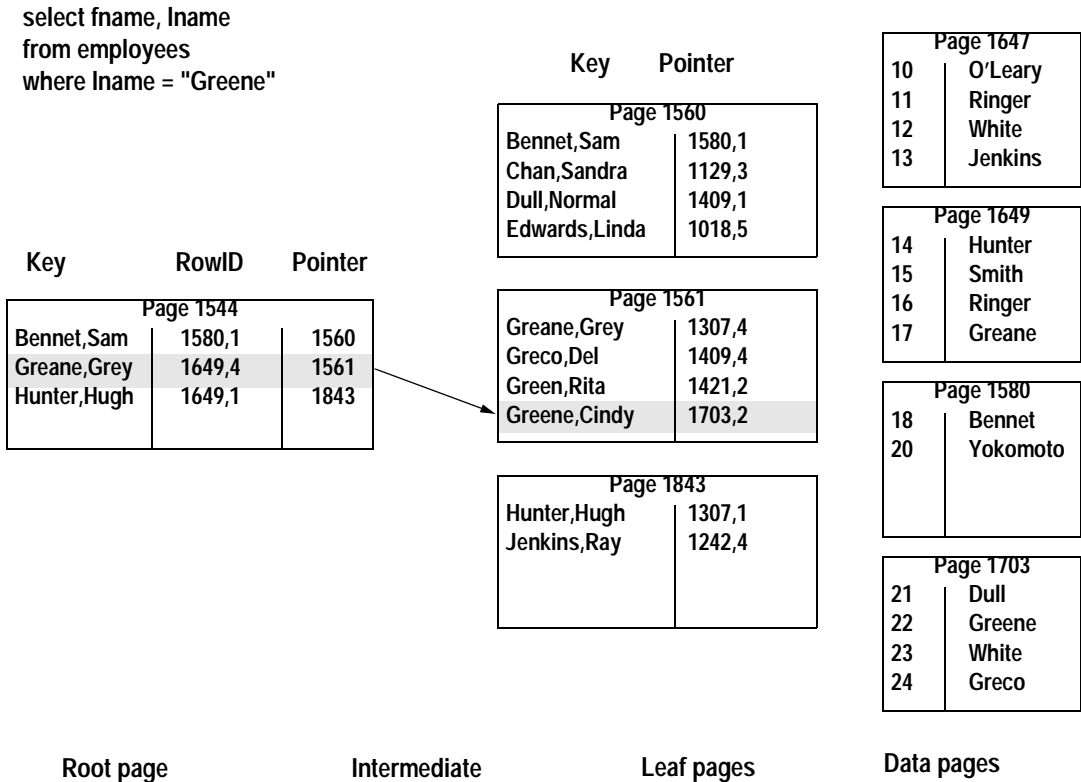
For a covering matching index scan to be used, the index must contain all columns named in the query. In addition, the columns in the where clauses of the query must include the leading column of the columns in the index.

For example, for an index on columns A, B, C, and D, the following sets can perform matching scans: A, AB, ABC, AC, ACD, ABD, AD, and ABCD. The columns B, BC, BCD, BD, C, CD, or D do not include the leading column and can be used only for nonmatching scans.

When doing a matching index scan, Adaptive Server uses standard index access methods to move from the root of the index to the nonclustered leaf page that contains the first row.

In Figure 12-11, the nonclustered index on lname, fname covers the query. The where clause includes the leading column, and all columns in the select list are included in the index, so the data page need not be accessed.

Figure 12-11: Matching index access does not have to read the data row



Covering nonmatching index scans

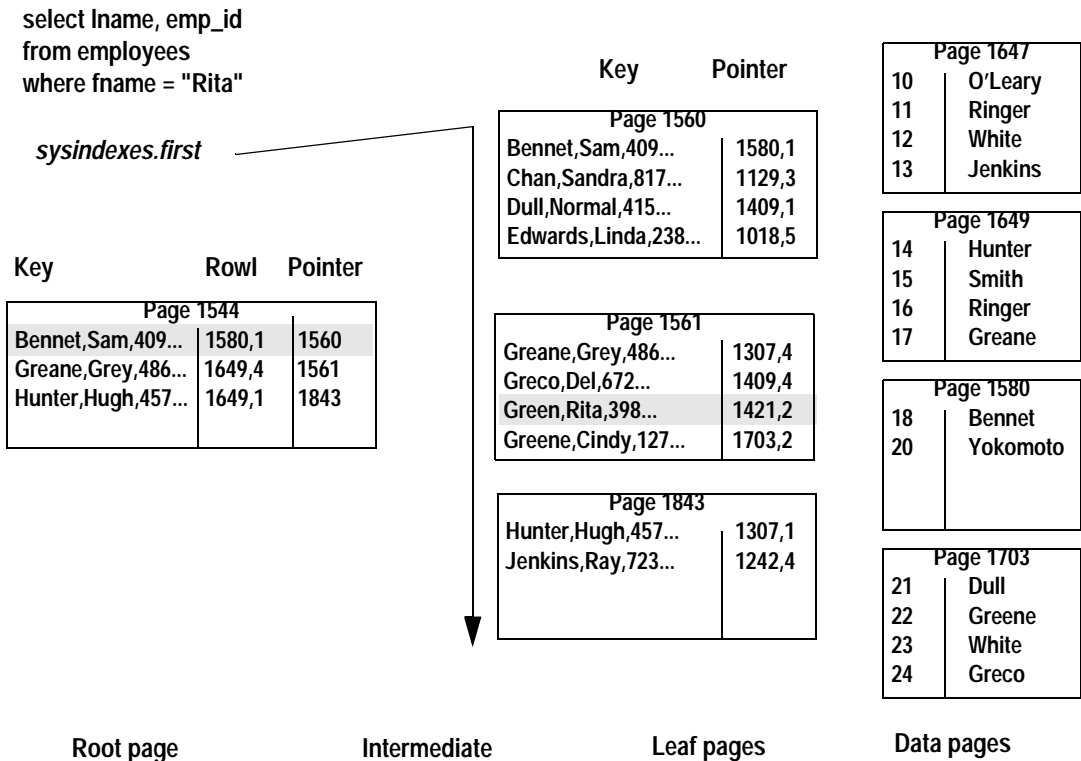
When the columns specified in the where clause do not include the leading column in the index, but all columns named in the select list and other query clauses (such as group by or having) are included in the index, Adaptive Server saves I/O by scanning the entire leaf level of the index, rather than scanning the table.

It cannot perform a matching scan because the first column of the index is not specified.

The query in Figure 12-12 shows a nonmatching index scan. This query does not use the leading columns on the index, but all columns required in the query are in the nonclustered index on lname, fname, emp_id.

The nonmatching scan must examine all rows on the leaf level. It scans all leaf level index pages, starting from the first page. It has no way of knowing how many rows might match the query conditions, so it must examine every row in the index. Since it must begin at the first page of the leaf level, it can use the pointer in sysindexes.first rather than descending the index.

Figure 12-12: A nonmatching index scan



Indexes and caching

“How Adaptive Server performs I/O for heap operations” on page 172 introduces the basic concepts of the Adaptive Server data cache, and shows how caches are used when reading heap tables.

Index pages get special handling in the data cache, as follows:

- Root and intermediate index pages always use LRU strategy.
- Index pages can use one cache while the data pages use a different cache, if the index is bound to a different cache.
- Covering index scans can use fetch-and-discard strategy.
- Index pages can cycle through the cache many times, if number of index trips is configured.

When a query that uses an index is executed, the root, intermediate, leaf, and data pages are read in that order. If these pages are not in cache, they are read into the MRU end of the cache and are moved toward the LRU end as additional pages are read in.

Each time a page is found in cache, it is moved to the MRU end of the page chain, so the root page and higher levels of the index tend to stay in the cache.

Using separate caches for data and index pages

Indexes and the tables they index can use different caches. A System Administrator or table owner can bind a clustered or nonclustered index to one cache and its table to another.

Index trips through the cache

A special strategy keeps index pages in cache. Data pages make only a single trip through the cache: they are read in at the MRU end of the cache or placed just before the wash marker, depending on the cache strategy chosen for the query.

Once the pages reach the LRU end of the cache, the buffer for that page is reused when another page needs to be read into cache.

For index pages, a counter controls the number of trips that an index page can make through the cache.

When the counter is greater than 0 for an index page, and it reaches the LRU end of the page chain, the counter is decremented by 1, and the page is placed at the MRU end again.

By default, the number of trips that an index page makes through the cache is set to 0. To change the default, a System Administrator can set the number of index trips configuration parameter

For more information, see the *System Administration Guide*.

Indexing for Performance

This chapter introduces the basic query analysis tools that can help you choose appropriate indexes and discusses index selection criteria for point queries, range queries, and joins.

Topic	Page
How indexes affect performance	297
Symptoms of poor indexing	298
Detecting indexing problems	298
Fixing corrupted indexes	301
Index limits and requirements	304
Choosing indexes	305
Techniques for choosing indexes	315
Index and statistics maintenance	317
Additional indexing tips	319

How indexes affect performance

Carefully considered indexes, built on top of a good database design, are the foundation of a high-performance Adaptive Server installation.

However, adding indexes without proper analysis can reduce the overall performance of your system. Insert, update, and delete operations can take longer when a large number of indexes need to be updated.

Analyze your application workload and create indexes as necessary to improve the performance of the most critical processes.

The Adaptive Server query optimizer uses a probabilistic costing model. It analyzes the costs of possible query plans and chooses the plan that has the lowest estimated cost. Since much of the cost of executing a query consists of disk I/O, creating the correct indexes for your applications means that the optimizer can use indexes to:

- Avoid table scans when accessing data

- Target specific data pages that contain specific values in a point query
- Establish upper and lower bounds for reading data in a range query
- Avoid data page access completely, when an index covers a query
- Use ordered data to avoid sorts or to favor merge joins over nested-loop joins

In addition, you can create indexes to enforce the uniqueness of data and to randomize the storage location of inserts.

Detecting indexing problems

Some of the major indications of insufficient or incorrect indexing include:

- A select statement takes too long.
- A join between two or more tables takes an extremely long time.
- Select operations perform well, but data modification processes perform poorly.
- Point queries (for example, “where colvalue = 3”) perform well, but range queries (for example, “where colvalue > 3 and colvalue < 30”) perform poorly.

These underlying problems are described in the following sections.

Symptoms of poor indexing

A primary goal of improving performance with indexes is avoiding table scans. In a table scan, every page of the table must be read from disk.

A query searching for a unique value in a table that has 600 data pages requires 600 physical and logical reads. If an index points to the data value, the same query can be satisfied with 2 or 3 reads, a performance improvement of 200 to 300 percent.

On a system with a 12-ms. disk, this is a difference of several seconds compared to less than a second. Heavy disk I/O by a single query has a negative impact on overall throughput.

Lack of indexes is causing table scans

If select operations and joins take too long, it probably indicates that either an appropriate index does not exist or, it exists, but is not being used by the optimizer.

showplan output reports whether the table is being accessed via a table scan or index. If you think that an index should be used, but showplan reports a table scan, dbcc traceon(302) output can help you determine the reason. It displays the costing computations for all optimizing query clauses.

If there is no clause is included in dbcc traceon(302) output, there may be problems with the way the clause is written. If a clause that you think should limit the scan is included in dbcc traceon(302) output, look carefully at its costing, and that of the chosen plan reported with dbcc traceon(310).

Index is not selective enough

An index is selective if it helps the optimizer find a particular row or a set of rows. An index on a unique identifier such as a Social Security Number is highly selective, since it lets the optimizer pinpoint a single row. An index on a nonunique entry such as sex (M, F) is not very selective, and the optimizer would use such an index only in very special cases.

Index does not support range queries

Generally, clustered indexes and covering indexes provide good performance for range queries and for search arguments (SARG) that match many rows. Range queries that reference the keys of noncovering indexes use the index for ranges that return a limited number of rows.

As the number of rows the query returns increases, however, using a nonclustered index or a clustered index on a data-only-locked table can cost more than a table scan.

Too many indexes slow data modification

If data modification performance is poor, you may have too many indexes. While indexes favor select operations, they slow down data modifications.

Every insert or delete operation affects the leaf level, (and sometimes higher levels) of a clustered index on a data-only-locked table, and each nonclustered index, for any locking scheme.

Updates to clustered index keys on allpages-locked tables can move the rows to different pages, requiring an update of every nonclustered index. Analyze the requirements for each index and try to eliminate those that are unnecessary or rarely used.

Index entries are too large

Try to keep index entries as small as possible. You can create indexes with keys up to 600 bytes, but those indexes can store very few rows per index page, which increases the amount of disk I/O needed during queries. The index has more levels, and each level has more pages.

The following example uses values reported by `sp_estspace` to demonstrate how the number of index pages and leaf levels required increases with key size. It creates nonclustered indexes using 10-, 20, and 40-character keys.

```
create table demotable (c10 char(10),
                      c20 char(20),
                      c40 char(40))
create index t10 on demotable(c10)
create index t20 on demotable(c20)
create index t40 on demotable(c40)
sp_estspace demotable, 500000
```

Table 13-1 shows the results.

Table 13-1: Effects of key size on index size and levels

Index, key size	Leaf-level pages	Index levels
t10, 10 bytes	4311	3
t20, 20 bytes	6946	3
t40, 40 bytes	12501	4

The output shows that the indexes for the 10-column and 20-column keys each have three levels, while the 40-column key requires a fourth level.

The number of pages required is more than 50 percent higher at each level.

Exception for wide data rows and wide index rows

Indexes with wide rows may be useful when:

- The table has very wide rows, resulting in very few rows per data page.
- The set of queries run on the table provides logical choices for a covering index.

- Queries return a sufficiently large number of rows.

For example, if a table has very long rows, and only one row per page, a query that needs to return 100 rows needs to access 100 data pages. An index that covers this query, even with long index rows, can improve performance.

For example, if the index rows were 240 bytes, the index would store 8 rows per page, and the query would need to access only 12 index pages.

Fixing corrupted indexes

If the index on one of your system tables has been corrupted, you can use the `sp_fixindex` system procedure to repair the index. For syntax information, see the entry for `sp_fixindex` in “System Procedures” in the *Adaptive Server Reference Manual*.

Repairing the system table index

Repairing a corrupted system table index requires the following steps:

❖ Repairing the system table index with `sp_fixindex`

- 1 Get the `object_name`, `object_ID`, and `index_ID` of the corrupted index. If you only have a page number and you need to find the `object_name`, see the *Adaptive Server Troubleshooting and Error Messages Guide* for instructions.
- 2 If the corrupted index is on a system table in the master database, put Adaptive Server in single-user mode. See the *Adaptive Server Troubleshooting and Error Messages Guide* for instructions.
- 3 If the corrupted index is on a system table in a user database, put the database in single-user mode and reconfigure to allow updates to system tables:

```
1> use master
2> go
1> sp_dboption database_name, "single user", true
2> go
1> sp_configure "allow updates", 1
2> go
```
- 4 Issue the `sp_fixindex` command:

```
1> use database_name
2> go

1> checkpoint
2> go

1> sp_fixindex database_name, object_name, index_ID
2> go
```

You can use the checkpoint to identify the one or more databasess or use an all clause.

```
checkpoint [all | [dbname[, dbname[, dbname.....]]]
```

Note You must possess sa_role permissions to run sp_fixindex.

5 Run dbcc checktable to verify that the corrupted index is now fixed.

6 Disallow updates to system tables:

```
1> use master
2> go

1> sp_configure "allow updates", 0
2> go
```

7 Turn off single-user mode:

```
1> sp_dboption database_name, "single user", false
2> go

1> use database_name
2> go

1> checkpoint
2> go
```

You can use the checkpoint to identify the one or more databasess or use an all clause, which means you do not have to issue the use database command.

```
checkpoint [all | [dbname[, dbname[, dbname.....]]]
```

Repairing a nonclustered index

Running sp_fixindex to repair a nonclustered index on sysobjects requires several additional steps.

❖ Repairing a nonclustered index on sysobjects

- 1 Perform steps 1-3, as described in “Repairing the system table index with sp_fixindex,” above.

- 2 Issue the following Transact-SQL query:

```
1> use database_name
2> go

1> checkpoint
2> go

1> select sysstat from sysobjects
2> where id = 1
3> go
```

You can use the checkpoint to identify the one or more databases or use an all clause.

```
checkpoint [all | [dbname[, dbname[, dbname.....]]]
```

- 3 Save the original sysstat value.
- 4 Change the sysstat column to the value required by sp_fixindex:

```
1> update sysobjects
2> set sysstat = sysstat | 4096
3> where id = 1
4> go
```

- 5 Run sp_fixindex:

```
1> sp_fixindex database_name, sysobjects, 2
2> go
```

- 6 Restore the original sysstat value:

```
1> update sysobjects
2> set sysstat = sysstat_ORIGINAL
3> where id = object_ID
4> go
```

- 7 Run dbcc checktable to verify that the corrupted index is now fixed.

- 8 Disallow updates to system tables:

```
1> sp_configure "allow updates", 0
2> go
```

- 9 Turn off single-user mode:

```
1> sp_dboption database_name, "single user", false
2> go
```

```
1> use database_name
2> go

1> checkpoint
2> go
```

You can use the checkpoint to identify the one or more databases or use an all clause.

```
checkpoint [all | [dbname[, dbname[, dbname.....]]]
```

Index limits and requirements

The following limits apply to indexes in Adaptive Server:

- You can create only one clustered index per table, since the data for a clustered index is ordered by index key.
- You can create a maximum of 249 nonclustered indexes per table.
- A key can be made up of as many as 31 columns. The maximum number of bytes per index key is 600.
- When you create a clustered index, Adaptive Server requires empty free space to copy the rows in the table and allocate space for the clustered index pages. It also requires space to re-create any nonclustered indexes on the table.

The amount of space required can vary, depending on how full the table's pages are when you begin and what space management properties are applied to the table and index pages.

See "Determining the space available for maintenance activities" on page 356 for more information.

- The referential integrity constraints unique and primary key create unique indexes to enforce their restrictions on the keys. By default, unique constraints create nonclustered indexes and primary key constraints create clustered indexes.

Choosing indexes

When you are working with index selection you may want to ask these questions:

- What indexes are associated currently with a given table?
- What are the most important processes that make use of the table?
- What is the ratio of select operations to data modifications performed on the table?
- Has a clustered index been created for the table?
- Can the clustered index be replaced by a nonclustered index?
- Do any of the indexes cover one or more of the critical queries?
- Is a composite index required to enforce the uniqueness of a compound primary key?
- What indexes can be defined as unique?
- What are the major sorting requirements?
- Do some queries use descending ordering of result sets?
- Do the indexes support joins and referential integrity checks?
- Does indexing affect update types (direct versus deferred)?
- What indexes are needed for cursor positioning?
- If dirty reads are required, are there unique indexes to support the scan?
- Should IDENTITY columns be added to tables and indexes to generate unique indexes? Unique indexes are required for updatable cursors and dirty reads.

When deciding how many indexes to use, consider:

- Space constraints
- Access paths to table
- Percentage of data modifications versus select operations
- Performance requirements of reports versus OLTP
- Performance impacts of index changes
- How often you can use update statistics

Index keys and logical keys

Index keys need to be differentiated from logical keys. Logical keys are part of the database design, defining the relationships between tables: primary keys, foreign keys, and common keys.

When you optimize your queries by creating indexes, the logical keys may or may not be used as the physical keys for creating indexes. You can create indexes on columns that are not logical keys, and you may have logical keys that are not used as index keys.

Choose index keys for performance reasons. Create indexes on columns that support the joins, search arguments, and ordering requirements in queries.

A common error is to create the clustered index for a table on the primary key, even though it is never used for range queries or ordering result sets.

Guidelines for clustered indexes

These are general guidelines for clustered indexes:

- Most allpages-locked tables should have clustered indexes or use partitions to reduce contention on the last page of heaps.

In a high-transaction environment, the locking on the last page severely limits throughput.

- If your environment requires a lot of inserts, do not place the clustered index key on a steadily increasing value such as an IDENTITY column.

Choose a key that places inserts on random pages to minimize lock contention while remaining useful in many queries. Often, the primary key does not meet this condition.

This problem is less severe on data-only-locked tables, but is a major source of lock contention on allpages-locked tables.

- Clustered indexes provide very good performance when the key matches the search argument in range queries, such as:

```
where colvalue >= 5 and colvalue < 10
```

In allpages-locked tables, rows are maintained in key order and pages are linked in order, providing very fast performance for queries using a clustered index.

In data-only-locked tables, rows are in key order after the index is created, but the clustering can decline over time.

- Other good choices for clustered index keys are columns used in order by clauses and in joins.
- If possible, do not include frequently updated columns as keys in clustered indexes on allpages-locked tables.

When the keys are updated, the rows must be moved from the current location to a new page. Also, if the index is clustered, but not unique, updates are done in deferred mode.

Choosing clustered indexes

Choose indexes based on the kinds of where clauses or joins you perform. Choices for clustered indexes are:

- The primary key, if it is used for where clauses and if it randomizes inserts
- Columns that are accessed by range, such as:

```
col11 between 100 and 200
col12 > 62 and < 70
```

- Columns used by order by
- Columns that are not frequently changed
- Columns used in joins

If there are several possible choices, choose the most commonly needed physical order as a first choice.

As a second choice, look for range queries. During performance testing, check for “hot spots” due to lock contention.

Candidates for nonclustered indexes

When choosing columns for nonclustered indexes, consider all the uses that were not satisfied by your clustered index choice. In addition, look at columns that can provide performance gains through index covering.

On data-only-locked tables, clustered indexes can perform index covering, since they have a leaf level above the data level.

On allpages-locked tables, noncovered range queries work well for clustered indexes, but may or may not be supported by nonclustered indexes, depending on the size of the range.

Consider using composite indexes to cover critical queries and to support less frequent queries:

- The most critical queries should be able to perform point queries and matching scans.
- Other queries should be able to perform nonmatching scans using the index, which avoids table scans.

Index Selection

Index selection allows you to determine which indexes are actively being used and those that are rarely used.

This section assumes that the monitoring tables feature is already set up, see *Performance and Tuning: Monitoring and Analyzing for Performance*, and includes the following steps:

- Add a 'loopback' server definition.
- Run `installmontables` to install the monitoring tables.
- Grant `mon_role` to all users who need to perform monitoring.
- Set the monitoring configuration parameters. For more information, see *Performance and Tuning: Monitoring and Analyzing for Performance*.

You can use `sp_monitorconfig` to track whether number of open objects or number of open indexes are sufficiently configured.

Index selection-usage uses the following five columns of the monitoring access table, `monOpenObjectActivity`:

- `IndexID` – unique identifier for the index.
- `OptSelectCount` – reports the number of times that the corresponding object (such as a table or index) was used as the access method by the optimizer.
- `LastOptSelectDate` – reports the last time `OptSelectCount` was incremented
- `UsedCount` – reports the number of times that the corresponding object (such as a table or index) was used as an access method when a query executed.
- `LastUsedDate` – reports the last time `UsedCount` was incremented.

If a plan has already been compiled and cached, `OptSelectCount` is not incremented each time the plan is executed. However, `UsedCount` is incremented when a plan is executed. If `no_exec` is on, `OptSelectCount` value is 3 incremented, but the `UsedCount` value does not.

Monitoring data is nonpersistent. That is, when you restart the server, the monitoring data is reset. Monitoring data is reported only for active objects. For example, monitoring data does not exist for objects that have not been opened since there are no active object descriptors for such objects. For systems that are inadequately configured and have reused object descriptors, monitoring data for these object descriptors is reinitialized and the data for the previous object is lost. When the old object is reopened, its monitoring data is reset.

Examples of using the index selection

The following example queries the monitoring tables for the last time all indexes for a specific object were selected by the optimizer as well as the last time they were actually used during execution, and reports the counts in each case:

```
select DBID, ObjectID, IndexID, OptSelectCount, LastOptSelectDate, UsedCount,
LastUsedDate
from monOpenObjectActivity
where DBID = db_id("financials_db") and ObjectID = object_id('expenses')
order by UsedCount
```

This example displays all indexes that are not currently used in an application or server:

```
select DBID , ObjectID, IndexID , object_name(ObjectID, DBID)
from monOpenObjectActivity
where DBID = db_id("financials_db") and OptSelectCount = 0
```

This example displays all indexes that are not currently used in an application, and also provides a sample output:

```
select DBID , ObjectID, IndexID , object_name(ObjectID, DBID)
from monOpenObjectActivity
where DBID = db_id("financials_db") and OptSelectCount = 0
ObjectName id IndexName OptCtLast OptSelectDate
UsedCount LastUsedDate
-----
-----
customer 2 ci_nkey_ckekey 3 Sep 27 2002 4:05PM
20 Sep 27 2002 4:05PM
customer 0 customer_x 3 Sep 27 2002 4:08PM
```

5	Sep 27 2002 4:08PM			
customer	1	customer_x	1	Sep 27 2002 4:06PM
5	Sep 27 2002 4:07PM			
customer	3	ci_ckey_nkey	1	Sep 27 2002 4:04PM
5	Sep 27 2002 4:05PM			
customer	4	customer_nation	0	Jan 1 1900 12:00AM
0	Jan 1 1900 12:00AM			

In this example, the `customer_nation` index has not been used, which results in the date “Jan 1 1900 12:00AM”.

Other indexing guidelines

Here are some other considerations for choosing indexes:

- If an index key is unique, define it as unique so the optimizer knows immediately that only one row matches a search argument or a join on the key.
- If your database design uses referential integrity (the `references` keyword or the `foreign key...references` keywords in the `create table` statement), the referenced columns *must* have a unique index, or the attempt to create the referential integrity constraint fails.

However, Adaptive Server does not automatically create an index on the referencing column. If your application updates primary keys or deletes rows from primary key tables, you may want to create an index on the referencing column so that these lookups do not perform a table scan.

- If your applications use cursors, see “Index use and requirements for cursors” on page 331.
- If you are creating an index on a table where there will be a lot of insert activity, use `fillfactor` to temporarily minimize page splits and improve concurrency and minimize deadlocking.
- If you are creating an index on a read-only table, use a `fillfactor` of 100 to make the table or index as compact as possible.
- Keep the size of the key as small as possible. Your index trees remain flatter, accelerating tree traversals.
- Use small datatypes whenever it fits your design.
 - Numerics compare slightly faster than strings internally.

- Variable-length character and binary types require more row overhead than fixed-length types, so if there is little difference between the average length of a column and the defined length, use fixed length. Character and binary types that accept null values are variable-length by definition.
- Whenever possible, use fixed-length, non-null types for short columns that will be used as index keys.
- Be sure that the datatypes of the join columns in different tables are compatible. If Adaptive Server has to convert a datatype on one side of a join, it may not use an index for that table.

See “Datatype mismatches and query optimization” on page 24 in *Performance and Tuning: Optimizer* for more information.

Choosing nonclustered indexes

When you consider adding nonclustered indexes, you must weigh the improvement in retrieval time against the increase in data modification time. In addition, you need to consider these questions:

- How much space will the indexes use?
- How volatile is the candidate column?
- How selective are the index keys? Would a scan be better?
- Are there a lot of duplicate values?

Because of data modification overhead, add nonclustered indexes only when your testing shows that they are helpful.

Performance price for data modification

Each nonclustered index needs to be updated, for all locking schemes:

- For each insert into the table
- For each delete from the table

An update to the table that changes part of an index’s key requires updating just that index.

For tables that use allpages locking, all indexes need to be updated:

- For any update that changes the location of a row by updating a clustered index key so that the row moves to another page
- For every row affected by a data page split

For allpages-locked tables, exclusive locks are held on affected index pages for the duration of the transaction, increasing lock contention as well as processing overhead.

Some applications experience unacceptable performance impacts with only three or four indexes on tables that experience heavy data modification. Other applications can perform well with many more tables.

Choosing composite indexes

If your analysis shows that more than one column is a good candidate for a clustered index key, you may be able to provide clustered-like access with a composite index that covers a particular query or set of queries. These include:

- Range queries.
- Vector (grouped) aggregates, if both the grouped and grouping columns are included. Any search arguments must also be included in the index.
- Queries that return a high number of duplicates.
- Queries that include order by.
- Queries that table scan, but use a small subset of the columns on the table.

Tables that are read-only or read-mostly can be heavily indexed, as long as your database has enough space available. If there is little update activity and high select activity, you should provide indexes for all of your frequent queries. Be sure to test the performance benefits of index covering.

Key order and performance in composite indexes

Covered queries can provide excellent response time for specific queries when the leading columns are used.

With the composite nonclustered index on `au_lname`, `au_fname`, `au_id`, this query runs very quickly:

```
select au_id
      from authors
 where au_fname = "Eliot" and au_lname = "Wilk"
```

This covered point query needs to read only the upper levels of the index and a single page in the leaf-level row in the nonclustered index of a 5000-row table.

This similar-looking query (using the same index) does not perform quite as well. This query is still covered, but searches on `au_id`:

```
select au_fname, au_lname
       from authors
       where au_id = "A1714224678"
```

Since this query does not include the leading column of the index, it has to scan the entire leaf level of the index, about 95 reads.

Adding a column to the select list in the query above, which may seem like a minor change, makes the performance even worse:

```
select au_fname, au_lname, phone
       from authors
       where au_id = "A1714224678"
```

This query performs a table scan, reading 222 pages. In this case, the performance is noticeably worse. For any search argument that is not the leading column, Adaptive Server has only two possible access methods: a table scan, or a covered index scan.

It does not scan the leaf level of the index for a non-leading search argument and then access the data pages. A composite index can be used only when it covers the query or when the first column appears in the where clause.

For a query that includes the leading column of the composite index, adding a column that is not included in the index adds only a single data page read. This query must read the data page to find the phone number:

```
select au_id, phone
       from authors
       where au_fname = "Eliot" and au_lname = "Wilk"
```

Table 13-2 shows the performance characteristics of different where clauses with a nonclustered index on `au_lname`, `au_fname`, `au_id` and no other indexes on the table.

Table 13-2: Composite nonclustered index ordering and performance

Columns in the where clause	Performance with the indexed columns in the select list	Performance with other columns in the select list
<code>au_lname</code>	Good; index used to descend tree; data level is not accessed	Good; index used to descend tree; data is accessed (one more page read per row)
<code>or au_lname, au_fname</code>		
<code>or au_lname, au_fname, au_id</code>		

Columns in the where clause	Performance with the indexed columns in the select list	Performance with other columns in the select list
au_fname or au_id or au_fname, au_id	Moderate; index is scanned to return values	Poor; index not used, table scan

Choose the ordering of the composite index so that most queries form a prefix subset.

Advantages and disadvantages of composite indexes

Composite indexes have these advantages:

- A composite index provides opportunities for index covering.
- If queries provide search arguments on each of the keys, the composite index requires fewer I/Os than the same query using an index on any single attribute.
- A composite index is a good way to enforce the uniqueness of multiple attributes.

Good choices for composite indexes are:

- Lookup tables
- Columns that are frequently accessed together
- Columns used for vector aggregates
- Columns that make a frequently used subset from a table with very wide rows

The disadvantages of composite indexes are:

- Composite indexes tend to have large entries. This means fewer index entries per index page and more index pages to read.
- An update to any attribute of a composite index causes the index to be modified. The columns you choose should not be those that are updated often.

Poor choices are:

- Indexes that are nearly as wide as the table
- Composite indexes where only a minor key is used in the where clause

Techniques for choosing indexes

This section presents a study of two queries that must access a single table, and the indexing choices for these two queries. The two queries are:

- A range query that returns a large number of rows
- A point query that returns only one or two rows

Choosing an index for a range query

Assume that you need to improve the performance of the following query:

```
select title
from titles
where price between $20.00 and $30.00
```

Some basic statistics on the table are:

- The table has 1,000,000 rows, and uses allpages locking.
- There are 10 rows per page; pages are 75 percent full, so the table has approximately 135,000 pages.
- 190,000 (19%) of the titles are priced between \$20 and \$30.

With no index, the query would scan all 135,000 pages.

With a clustered index on price, the query would find the first \$20 book and begin reading sequentially until it gets to the last \$30 book. With pages about 75 percent full, the average number of rows per page is 7.5. To read 190,000 matching rows, the query would read approximately 25,300 pages, plus 3 or 4 index pages.

With a nonclustered index on price and random distribution of price values, using the index to find the rows for this query requires reading about 19 percent of the leaf level of the index, about 1,500 pages.

If the price values are randomly distributed, the number of data pages that must be read is likely to be high, perhaps as many data pages as there are qualifying rows, 190,000. Since a table scan requires only 135,000 pages, you would not want to use this nonclustered.

Another choice is a nonclustered index on price, title. The query can perform a matching index scan, using the index to find the first page with a price of \$20, and then scanning forward on the leaf level until it finds a price of more than \$30. This index requires about 35,700 leaf pages, so to scan the matching leaf pages requires reading about 19 percent of the pages of this index, or about 6,800 reads.

For this query, the covering nonclustered index on price, title is best.

Adding a point query with different indexing requirements

The index choice for the range query on price produced a clear performance choice when all possibly useful indexes were considered. Now, assume this query also needs to run against titles:

```
select price
from titles
where title = "Looking at Leeks"
```

You know that there are very few duplicate titles, so this query returns only one or two rows.

Considering both this query and the previous query, Table 13-3 shows four possible indexing strategies and estimate costs of using each index. The estimates for the numbers of index and data pages were generated using a fillfactor of 75 percent with sp_estspace:

```
sp_estspace titles, 1000000, 75
```

The values were rounded for easier comparison.

Table 13-3: Comparing index strategies for two queries

Possible index choice	Index pages	Range query on price	Point query on title
1 Nonclustered on title Clustered on price	36,800 650	Clustered index, about 26,600 pages (135,000 *.19) With 16K I/O: 3,125 I/Os	Nonclustered index, 6 I/Os
2 Clustered on title Nonclustered on price	3,770 6,076	Table scan, 135,000 pages With 16K I/O: 17,500 I/Os	Clustered index, 6 I/Os
3 Nonclustered on title, price	36,835	Nonmatching index scan, about 35,700 pages With 16K I/O: 4,500 I/Os	Nonclustered index, 5 I/Os
4 Nonclustered on price, title	36,835	Matching index scan, about 6,800 pages (35,700 *.19) With 16K I/O: 850 I/Os	Nonmatching index scan, about 35,700 pages With 16K I/O: 4,500 I/Os

Examining the figures in Table 13-3 shows that:

- For the range query on price, choice 4 is best; choices 1 and 3 are acceptable with 16K I/O.
- For the point query on titles, indexing choices 1, 2, and 3 are excellent.

The best indexing strategy for a combination of these two queries is to use two indexes:

- Choice 4, for range queries on price.
- Choice 2, for point queries on title, since the clustered index requires very little space.

You may need additional information to help you determine which indexing strategy to use to support multiple queries. Typical considerations are:

- What is the frequency of each query? How many times per day or per hour is the query run?
- What are the response time requirements? Is one of them especially time critical?
- What are the response time requirements for updates? Does creating more than one index slow updates?
- Is the range of values typical? Is a wider or narrower range of prices, such as \$20 to \$50, often used? How do different ranges affect index choice?
- Is there a large data cache? Are these queries critical enough to provide a 35,000-page cache for the nonclustered composite indexes in index choice 3 or 4? Binding this index to its own cache would provide very fast performance.
- What other queries and what other search arguments are used? Is this table frequently joined with other tables?

Index and statistics maintenance

To ensure that indexes evolve with your system:

- Monitor queries to determine if indexes are still appropriate for your applications.

Periodically, check the query plans, as described in Chapter 5, “Using set showplan,” in the *Performance and Tuning: Monitoring and Analyzing for Performance* book and the I/O statistics for your most frequent user queries. Pay special attention to noncovering indexes that support range queries. They are most likely to switch to table scans if the data distribution changes

- Drop and rebuild indexes that hurt performance.
- Keep index statistics up to date.
- Use space management properties to reduce page splits and to reduce the frequency of maintenance operations.

Dropping indexes that hurt performance

Drop indexes that hurt performance. If an application performs data modifications during the day and generates reports at night, you may want to drop some indexes in the morning and re-create them at night.

Many system designers create numerous indexes that are rarely, if ever, actually used by the query optimizer. Make sure that you base indexes on the current transactions and processes that are being run, not on the original database design.

Check query plans to determine whether your indexes are being used.

For more information on maintaining indexes see “Maintaining index and column statistics” on page 346 and “Rebuilding indexes” on page 347.

Choosing space management properties for indexes

Space management properties can help reduce the frequency of index maintenance. In particular, fillfactor can reduce the number of page splits on leaf pages of nonclustered indexes and on the data pages of allpages-locked tables with clustered indexes.

See Chapter 9, “Setting Space Management Properties,” for more information on choosing fillfactor values for indexes.

Additional indexing tips

Here are some additional suggestions that can lead to improved performance when you are creating and using indexes:

- Modify the logical design to make use of an artificial column and a lookup table for tables that require a large index entry.
- Reduce the size of an index entry for a frequently used index.
- Drop indexes during periods when frequent updates occur and rebuild them before periods when frequent selects occur.
- If you do frequent index maintenance, configure your server to speed up the sorting.

See “Configuring Adaptive Server to speed sorting” on page 344 for information about configuration parameters that enable faster sorting.

Creating artificial columns

When indexes become too large, especially composite indexes, it is beneficial to create an artificial column that is assigned to a row, with a secondary lookup table that is used to translate between the internal ID and the original columns.

This may increase response time for certain queries, but the overall performance gain due to a more compact index and shorter data rows is usually worth the effort.

Keeping index entries short and avoiding overhead

Avoid storing purely numeric IDs as character data. Use integer or numeric IDs whenever possible to:

- Save storage space on the data pages
- Make index entries more compact
- Improve performance, since internal comparisons are faster

Index entries on varchar columns require more overhead than entries on char columns. For short index keys, especially those with little variation in length in the column data, use char for more compact index entries.

Dropping and rebuilding indexes

You might drop nonclustered indexes prior to a major set of inserts, and then rebuild them afterwards. In that way, the inserts and bulk copies go faster, since the nonclustered indexes do not have to be updated with every insert.

For more information, see “Rebuilding indexes” on page 347.

Configure enough sort buffers

The sort buffers decides how many pages of data you can sort in each run. That is the basis for the logarithmic function on calculating the number of runs needed to finish the sort.

For example, if you have 500 buffers, then the number of runs is calculated with "log (number of pages in table) with 500 as the log base".

Also note that the number of sort buffers is shared by threads in the parallel sort, if you do not have enough sort buffers, the parallel sort may not work as fast as it should.

Create the clustered index first

Do not create nonclustered indexes, then clustered indexes. When you create the clustered index all previous nonclustered indexes are rebuilt.

Configure large buffer pools

To set up for larger O/Os, configure large buffers pools in a named cache and bind the cache to the table.

Asynchronous log service

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

You cannot use ALS if you have fewer than 4 engines. If you try to enable ALS with fewer than 4 online engines an error message appears.

Enabling ALS

You can enable, disable, or configure ALS using the `sp_dboption` stored procedure.

```
sp_dboption <db Name>, "async log service",
"true|false"
```

Issuing a checkpoint

After issuing `sp_dboption`, you must issue a checkpoint in the database for which you are setting the ALS option:

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

You can use the checkpoint to identify the one or more databases or use an `all` clause.

```
checkpoint [all | [dbname[, dbname[, dbname.....]]]
```

Disabling ALS

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

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

```
-----
```

```
Error 3647: Cannot put database in single-user mode.
Wait until all users have logged out of the database and
issue a CHECKPOINT to disable "async log service".
```

If there are no active users in the database, this example disables ALS:

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

```
-----
```

Displaying ALS

You can see whether ALS is enabled in a specified database by checking `sp_helpdb`.

```
sp_helpdb "mydb"
```

```
-----
```

```
mydb          3.0 MB sa          2
              July 09, 2002
              select into/bulkcopy/pllsort, trunc log on chkpt,
              async log service
```

Understanding the user log cache (ULC) architecture

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

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

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

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

- 4 Flush the log cache to disk.

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

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

When to use ALS

You can enable ALS on any specified database that has at least one of the following performance issues, so long as your systems runs 4 or more online engines:

- Heavy contention on the last log page.

You can tell that the last log page is under contention when the `sp_sysmon` output in the Task Management Report section shows a significantly high value. For example:

Table 13-4: Log page under contention

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

- Heavy contention on the cache manager spinlock for the log cache.

You can tell that the cache manager spinlock is under contention when the `sp_sysmon` output in the Data Cache Management Report section for the database transaction log cache shows a high value in the Spinlock Contention section. For example:

Table 13-5:

Cache c_log	per sec	per xact	count	% of total
Spinlock Contention	n/a	n/a	n/a	40.0%

- Underutilized bandwidth in the log device.

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

Using the ALS

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

- The User Log Cache (ULC) flusher
- The Log Writer.

ULC flusher

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

Log writer

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

Changes in stored procedures

Asynchronous log service changes the stored procedures `sp_dboption` and `sp_helpdb`:

- `sp_dboption` adds an option that enables and disables ALS.
- `sp_helpdb` adds a column to display ALS.

For more information on `sp_helpdb` and `sp_dboption`, see the *Reference Manual*.

This chapter discusses performance issues related to cursors. Cursors are a mechanism for accessing the results of a SQL `select` statement one row at a time (or several rows, if you use set cursors rows). Since cursors use a different model from ordinary set-oriented SQL, the way cursors use memory and hold locks has performance implications for your applications. In particular, cursor performance issues includes locking at the page and at the table level, network resources, and overhead of processing instructions.

Topic	Page
Definition	325
Resources required at each stage	328
Cursor modes	331
Index use and requirements for cursors	331
Comparing performance with and without cursors	333
Locking with read-only cursors	336
Isolation levels and cursors	338
Partitioned heap tables and cursors	338
Optimizing tips for cursors	339

Definition

A cursor is a symbolic name that is associated with a `select` statement. It enables you to access the results of a `select` statement one row at a time. Figure 14-1 shows a cursor accessing the `authors` table.

Figure 14-1: Cursor example

Cursor with select * from authors where state = 'KY'		Result set			
	➔	A978606525	Marcello	Duncan	KY
	➔	A937406538	Carton	Nita	KY
	➔	A1525070956	Porczyk	Howard	KY
Programming can: - Examine a row - Take an action based on row values	➔	A913907285	Bier	Lane	KY

You can think of a cursor as a “handle” on the result set of a select statement. It enables you to examine and possibly manipulate one row at a time.

Set-oriented versus row-oriented programming

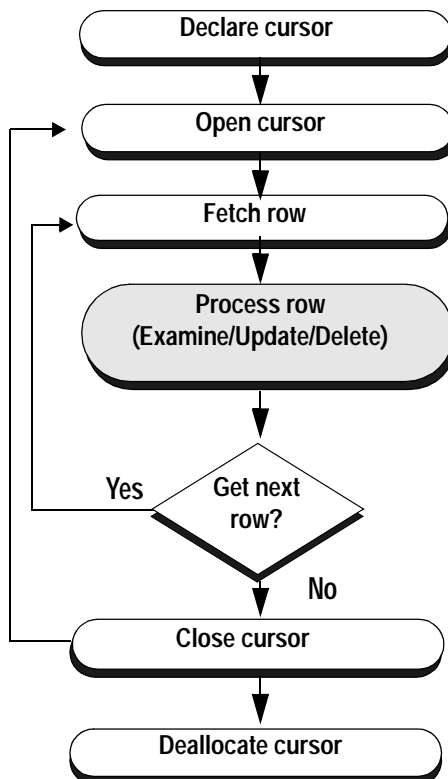
SQL was conceived as a set-oriented language. Adaptive Server is extremely efficient when it works in set-oriented mode. Cursors are required by ANSI SQL standards; when they are needed, they are very powerful. However, they can have a negative effect on performance.

For example, this query performs the identical action on all rows that match the condition in the where clause:

```
update titles
  set contract = 1
  where type = 'business'
```

The optimizer finds the most efficient way to perform the update. In contrast, a cursor would examine each row and perform single-row updates if the conditions were met. The application declares a cursor for a select statement, opens the cursor, fetches a row, processes it, goes to the next row, and so forth. The application may perform quite different operations depending on the values in the current row, and the server’s overall use of resources for the cursor application may be less efficient than the server’s set level operations. However, cursors can provide more flexibility than set-oriented programming.

Figure 14-2 shows the steps involved in using cursors. The function of cursors is to get to the middle box, where the user or application code examines a row and decides what to do, based on its values.

Figure 14-2: Cursor flowchart

Example

Here is a simple example of a cursor with the “Process Rows” step shown above in pseudocode:

```
declare biz_book cursor
  for select * from titles
    where type = 'business'
go
open biz_book
go
fetch biz_book
go
/* Look at each row in turn and perform
** various tasks based on values,
```

```
** and repeat fetches, until  
** there are no more rows  
*/  
close biz_book  
go  
deallocate cursor biz_book  
go
```

Depending on the content of the row, the user might delete the current row:

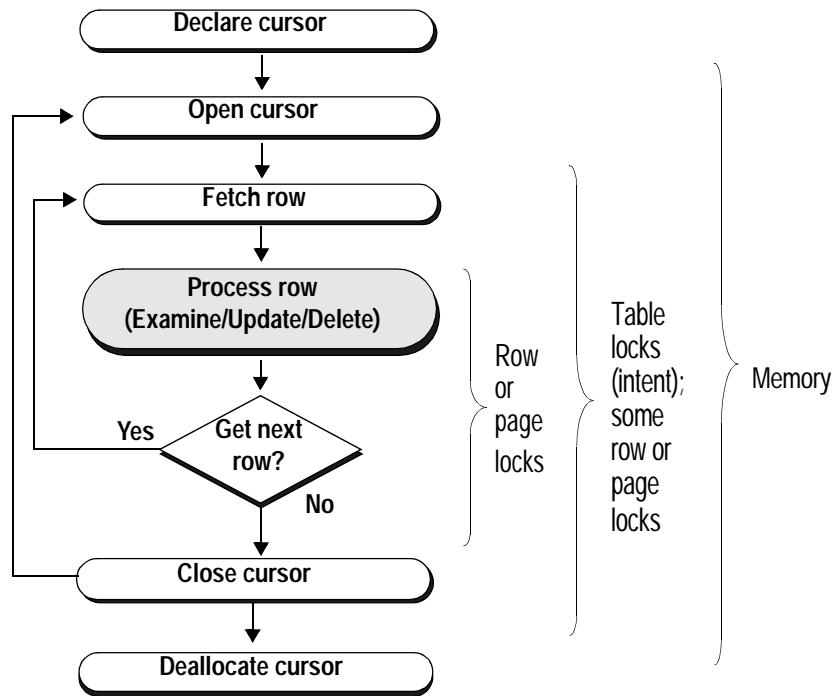
```
delete titles where current of biz_book
```

or update the current row:

```
update titles set title="The Rich  
Executive's Database Guide"  
where current of biz_book
```

Resources required at each stage

Cursors use memory and require locks on tables, data pages, and index pages. When you open a cursor, memory is allocated to the cursor and to store the query plan that is generated. While the cursor is open, Adaptive Server holds intent table locks and sometimes row or page locks. Figure 14-3 shows the duration of locks during cursor operations.

Figure 14-3: Resource use by cursor statement

The memory resource descriptions in Figure 14-3 and Table 14-1 refer to ad hoc cursors for queries sent by `isql` or `Client-Library™`. For other kinds of cursors, the locks are the same, but the memory allocation and deallocation differ somewhat depending on the type of cursor being used, as described in “Memory use and execute cursors” on page 330.

Table 14-1: Locks and memory use for isql and Client-Library client cursors

Cursor command	Resource use
declare cursor	When you declare a cursor, Adaptive Server uses only enough memory to store the query text.
open	When you open a cursor, Adaptive Server allocates memory to the cursor and to store the query plan that is generated. The server optimizes the query, traverses indexes, and sets up memory variables. The server does not access rows yet, unless it needs to build worktables. However, it does set up the required table-level locks (intent locks). Row and page locking behavior depends on the isolation level, server configuration, and query type. See <i>System Administration Guide</i> for more information.
fetch	When you execute a fetch, Adaptive Server gets the row(s) required and reads specified values into the cursor variables or sends the row to the client. If the cursor needs to hold lock on rows or pages, the locks are held until a fetch moves the cursor off the row or page or until the cursor is closed. The lock is either a shared or an update lock, depending on how the cursor is written.
close	When you close a cursor, Adaptive Server releases the locks and some of the memory allocation. You can open the cursor again, if necessary.
deallocate cursor	When you deallocate a cursor, Adaptive Server releases the rest of the memory resources used by the cursor. To reuse the cursor, you must declare it again.

Memory use and execute cursors

The descriptions of declare cursor and deallocate cursor in Table 14-1 refer to ad hoc cursors that are sent by isql or Client-Library. Other kinds of cursors allocate memory differently:

- For cursors that are declared *on* stored procedures, only a small amount of memory is allocated at declare cursor time. Cursors declared on stored procedures are sent using Client-Library or the precompiler and are known as execute cursors.
- For cursors declared *within* a stored procedure, memory is already available for the stored procedure, and the declare statement does not require additional memory.

Cursor modes

There are two cursor modes: read-only and update. As the names suggest, read-only cursors can only display data from a select statement; update cursors can be used to perform positioned updates and deletes.

Read-only mode uses shared page or row locks. If read committed with lock is set to 0, and the query runs at isolation level 1, it uses instant duration locks, and does not hold the page or row locks until the next fetch.

Read-only mode is in effect when you specify for read only or when the cursor's select statement uses distinct, group by, union, or aggregate functions, and in some cases, an order by clause.

Update mode uses update page or row locks. It is in effect when:

- You specify for update.
- The select statement does not include distinct, group by, union, a subquery, aggregate functions, or the at isolation read uncommitted clause.
- You specify shared.

If *column_name_list* is specified, only those columns are updatable.

For more information on locking during cursor processing, see *System Administration Guide*.

Specify the cursor mode when you declare the cursor. If the select statement includes certain options, the cursor is not updatable even if you declare it for update.

Index use and requirements for cursors

When a query is used in a cursor, it may require or choose different indexes than the same query used outside of a cursor.

Allpages-locked tables

For read-only cursors, queries at isolation level 0 (dirty reads) require a unique index. Read-only cursors at isolation level 1 or 3 should produce the same query plan as the select statement outside of a cursor.

The index requirements for updatable cursors mean that updatable cursors may use different query plans than read-only cursors. Update cursors have these indexing requirements:

- If the cursor is not declared for update, a unique index is preferred over a table scan or a nonunique index.
- If the cursor is declared for update *without* a for update of list, a unique index is required on allpages-locked tables. An error is raised if no unique index exists.
- If the cursor is declared for update with a for update of list, then only a unique index *without* any columns from the list can be chosen on an allpages-locked table. An error is raised if no unique index qualifies.

When cursors are involved, an index that contains an IDENTITY column is considered unique, even if the index is not declared unique. In some cases, IDENTITY columns must be added to indexes to make them unique, or the optimizer might be forced to choose a suboptimal query plan for a cursor query.

Data-only-locked tables

In data-only-locked tables, fixed row IDs are used to position cursor scans, so unique indexes are not required for dirty reads or updatable cursors. The only cause for different query plans in updatable cursors is that table scans are used if columns from only useful indexes are included in the for update of list.

Table scans to avoid the Halloween problem

The Halloween problem is an update anomaly that can occur when a client using a cursor updates a column of the cursor result-set row, and that column defines the order in which the rows are returned from the table. For example, if a cursor was to use an index on last_name, first_name, and update one of these columns, the row could appear in the result set a second time.

To avoid the Halloween problem on data-only-locked tables, Adaptive Server chooses a table scan when the columns from an otherwise useful index are included in the column list of a for update clause.

For implicitly updatable cursors declared without a for update clause, and for cursors where the column list in the for update clause is empty, cursors that update a column in the index used by the cursor may encounter the Halloween problem.

Comparing performance with and without cursors

This section examines the performance of a stored procedure written two different ways:

- Without a cursor – this procedure scans the table three times, changing the price of each book.
- With a cursor – this procedure makes only one pass through the table.

In both examples, there is a unique index on titles(title_id).

Sample stored procedure without a cursor

This is an example of a stored procedure without cursors:

```

/* Increase the prices of books in the
** titles table as follows:
**
** If current price is <= $30, increase it by 20%
** If current price is > $30 and <= $60, increase
** it by 10%
** If current price is > $60, increase it by 5%
**
** All price changes must take effect, so this is
** done in a single transaction.
*/

create procedure increase_price
as

    /* start the transaction */
    begin transaction
    /* first update prices > $60 */
    update titles
        set price = price * 1.05
        where price > $60

    /* next, prices between $30 and $60 */
    update titles
        set price = price * 1.10
    where price > $30 and price <= $60

    /* and finally prices <= $30 */
    update titles
        set price = price * 1.20

```

```
        where price <= $30

        /* commit the transaction */
        commit transaction

return
```

Sample stored procedure with a cursor

This procedure performs the same changes to the underlying table as the procedure written without a cursor, but it uses cursors instead of set-oriented programming. As each row is fetched, examined, and updated, a lock is held on the appropriate data page. Also, as the comments indicate, each update commits as it is made, since there is no explicit transaction.

```
/* Same as previous example, this time using a
** cursor. Each update commits as it is made.
*/
create procedure increase_price_cursor
as
declare @price money

/* declare a cursor for the select from titles */
declare curs cursor for
    select price
    from titles
    for update of price

/* open the cursor */
open curs

/* fetch the first row */
fetch curs into @price

/* now loop, processing all the rows
** @@sqlstatus = 0 means successful fetch
** @@sqlstatus = 1 means error on previous fetch
** @@sqlstatus = 2 means end of result set reached
*/
while (@@sqlstatus != 2)
begin
    /* check for errors */
    if (@@sqlstatus = 1)
    begin
        print "Error in increase_price"
```

```
        return
    end

    /* next adjust the price according to the
    ** criteria
    */
    if @price > $60
        select @price = @price * 1.05
    else
        if @price > $30 and @price <= $60
            select @price = @price * 1.10
        else
            if @price <= $30
                select @price = @price * 1.20
            end
        end
    end

    /* now, update the row */
    update titles
    set price = @price
    where current of curs

    /* fetch the next row */
    fetch curs into @price
end

/* close the cursor and return */
close curs
return
```

Which procedure do you think will have better performance, one that performs three table scans or one that performs a single scan via a cursor?

Cursor versus noncursor performance comparison

Table 14-2 shows statistics gathered against a 5000-row table. The cursor code takes over 4 times longer, even though it scans the table only once.

Table 14-2: Sample execution times against a 5000-row table

Procedure	Access method	Time
increase_price	Uses three table scans	28 seconds
increase_price_cursor	Uses cursor, single table scan	125 seconds

Results from tests like these can vary widely. They are most pronounced on systems that have busy networks, a large number of active database users, and multiple users accessing the same table.

In addition to locking, cursors involve more network activity than set operations and incur the overhead of processing instructions. The application program needs to communicate with Adaptive Server regarding every result row of the query. This is why the cursor code took much longer to complete than the code that scanned the table three times.

Cursor performance issues include:

- Locking at the page and table level
- Network resources
- Overhead of processing instructions

If there is a set-level programming equivalent, it may be preferable, even if it involves multiple table scans.

Locking with read-only cursors

Here is a piece of cursor code you can use to display the locks that are set up at each point in the life of a cursor. The following example uses an allpages-locked table. Execute the code in Figure 14-4, and pause at the arrows to execute `sp_lock` and examine the locks that are in place.

Figure 14-4: Read-only cursors and locking experiment input

```

declare curs1 cursor for
select au_id, au_fname, au_fname
  from authors
  where au_id like '15%'
  for read only
go
open curs1
go
fetch curs1
go
fetch curs1
go 100
close curs1
go
deallocate cursor curs1
go

```

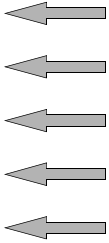


Table 14-3 shows the results.

Table 14-3: Locks held on data and index pages by cursors

Event	Data page
After declare	No cursor-related locks.
After open	Shared intent lock on authors.
After first fetch	Shared intent lock on authors and shared page lock on a page in authors.
After 100 fetches	Shared intent lock on authors and shared page lock on a different page in authors.
After close	No cursor-related locks.

If you issue another fetch command after the last row of the result set has been fetched, the locks on the last page are released, so there will be no cursor-related locks.

With a data-only-locked table:

- If the cursor query runs at isolation level 1, and read committed with lock is set to 0, you do not see any page or row locks. The values are copied from the page or row, and the lock is immediately released.
- If read committed with lock is set to 1 or if the query runs at isolation level 2 or 3, you see either shared page or shared row locks at the point that Table 14-3 indicates shared page locks. If the table uses datarows locking, the `sp_lock` report includes the row ID of the fetched row.

Isolation levels and cursors

The query plan for a cursor is compiled and optimized when the cursor is opened. You cannot open a cursor and then use set transaction isolation level to change the isolation level at which the cursor operates.

Since cursors using isolation level 0 are compiled differently from those using other isolation levels, you cannot open a cursor at isolation level 0 and open or fetch from it at level 1 or 3. Similarly, you cannot open a cursor at level 1 or 3 and then fetch from it at level 0. Attempts to fetch from a cursor at an incompatible level result in an error message.

Once the cursor has been opened at a particular isolation level, you must deallocate the cursor before changing isolation levels. The effects of changing isolation levels while the cursor is open are as follows:

- Attempting to close and reopen the cursor at another isolation level fails with an error message.
- Attempting to change isolation levels without closing and reopening the cursor has no effect on the isolation level in use and does not produce an error message.

You can include an at isolation clause in the cursor to specify an isolation level. The cursor in the example below can be declared at level 1 and fetched from level 0 because the query plan is compatible with the isolation level:

```
declare cprice cursor for
select title_id, price
  from titles
 where type = "business"
  at isolation read uncommitted
```

Partitioned heap tables and cursors

A cursor scan of an unpartitioned heap table can read all data up to and including the final insertion made to that table, even if insertions took place after the cursor scan started.

If a heap table is partitioned, data can be inserted into one of the many page chains. The physical insertion point may be before or after the current position of a cursor scan. This means that a cursor scan against a partitioned table is *not* guaranteed to scan the final insertions made to that table.

Note If your cursor operations require all inserts to be made at the end of a single page chain, *do not* partition the table used in the cursor scan.

Optimizing tips for cursors

Here are several optimizing tips for cursors:

- Optimize cursor selects using the cursor, not an ad hoc query.
- Use union or union all instead of or clauses or in lists.
- Declare the cursor's intent.
- Specify column names in the for update clause.
- Fetch more than one row if you are returning rows to the client.
- Keep cursors open across commits and rollbacks.
- Open multiple cursors on a single connection.

Optimizing for cursor selects using a cursor

A standalone select statement may be optimized very differently than the same select statement in an implicitly or explicitly updatable cursor. When you are developing applications that use cursors, always check your query plans and I/O statistics using the cursor, rather than using a standalone select. In particular, index restrictions of updatable cursors require very different access methods.

Using *union* instead of *or* clauses or *in* lists

Cursors cannot use the dynamic index of row IDs generated by the OR strategy. Queries that use the OR strategy in standalone select statements usually perform table scans using read-only cursors. Updatable cursors may need to use a unique index and still require access to each data row, in sequence, in order to evaluate the query clauses.

See “Access Methods and Costing for or and in Clauses” on page 87 in the book *Performance and Tuning: Optimizer* for more information.

A read-only cursor using union creates a worktable when the cursor is declared, and sorts it to remove duplicates. Fetches are performed on the worktable. A cursor using union all can return duplicates and does not require a worktable.

Declaring the cursor’s intent

Always declare a cursor’s intent: read-only or updatable. This gives you greater control over concurrency implications. If you do not specify the intent, Adaptive Server decides for you, and very often it chooses updatable cursors. Updatable cursors use update locks, thereby preventing other update locks or exclusive locks. If the update changes an indexed column, the optimizer may need to choose a table scan for the query, resulting in potentially difficult concurrency problems. Be sure to examine the query plans for queries that use updatable cursors.

Specifying column names in the *for update* clause

Adaptive Server acquires update locks on the pages or rows of all tables that have columns listed in the for update clause of the cursor select statement. If the for update clause is not included in the cursor declaration, all tables referenced in the from clause acquire update locks.

The following query includes the name of the column in the for update clause, but acquires update locks only on the titles table, since price is mentioned in the for update clause. The table uses allpages locking. The locks on authors and titleauthor are shared page locks:

```
declare curs3 cursor
for
select au_lname, au_fname, price
   from titles t, authors a,
      titleauthor ta
```

```

where advance <= $1000
      and t.title_id = ta.title_id
      and a.au_id = ta.au_id
for update of price

```

Table 14-4 shows the effects of:

- Omitting the for update clause entirely—no shared clause
- Omitting the column name from the for update clause
- Including the name of the column to be updated in the for update clause
- Adding shared after the name of the titles table while using for update of price

In this table, the additional locks, or more restrictive locks for the two versions of the for update clause are emphasized.

Table 14-4: Effects of for update clause and shared on cursor locking

Clause	<i>titles</i>	<i>authors</i>	<i>titleauthor</i>
None		sh_page on index	
	sh_page on data	sh_page on data	sh_page on data
for update	<i>updpage on index</i>	<i>updpage on index</i>	
	updpage on data	<i>updpage on data</i>	<i>updpage on data</i>
for update of price		sh_page on index	
	updpage on data	sh_page on data	sh_page on data
for update of price + shared		sh_page on index	
	sh_page on data	sh_page on data	sh_page on data

Using *set cursor rows*

The SQL standard specifies a one-row fetch for cursors, which wastes network bandwidth. Using the set cursor rows query option and Open Client’s transparent buffering of fetches, you can improve performance:

```

ct_cursor ( CT_CURSOR_ROWS )

```

Be careful when you choose the number of rows returned for frequently executed applications using cursors—tune them to the network.

See “Changing network packet sizes” on page 27 for an explanation of this process.

Keeping cursors open across commits and rollbacks

ANSI closes cursors at the conclusion of each transaction. Transact-SQL provides the `set option close on endtran` for applications that must meet ANSI behavior. By default, however, this option is turned off. Unless you must meet ANSI requirements, leave this option off to maintain concurrency and throughput.

If you must be ANSI-compliant, decide how to handle the effects on Adaptive Server. Should you perform a lot of updates or deletes in a single transaction? Or should you keep the transactions short?

If you choose to keep transactions short, closing and opening the cursor can affect throughput, since Adaptive Server needs to rematerialize the result set each time the cursor is opened. Choosing to perform more work in each transaction, this can cause concurrency problems, since the query holds locks.

Opening multiple cursors on a single connection

Some developers simulate cursors by using two or more connections from DB-Library™. One connection performs a select and the other performs updates or deletes on the same tables. This has very high potential to create application deadlocks. For example:

- Connection A holds a shared lock on a page. As long as there are rows pending from Adaptive Server, a shared lock is kept on the current page.
- Connection B requests an exclusive lock on the same pages and then waits.
- The application waits for Connection B to succeed before invoking whatever logic is needed to remove the shared lock. But this never happens.

Since Connection A never requests a lock that is held by Connection B, this is not a server-side deadlock.

Maintenance Activities and Performance

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

Maintenance activities include such tasks as dropping and re-creating indexes, performing dbcc checks, and updating 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 what kind of performance impacts these maintenance activities have on applications and on overall Adaptive Server performance.

Topic	Page
Running reorg on tables and indexes	343
Creating and maintaining indexes	344
Creating or altering a database	348
Backup and recovery	350
Bulk copy	352
Database consistency checker	355
Using dbcc tune (cleanup)	355
Using dbcc tune on spinlocks	356
Determining the space available for maintenance activities	356

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 space 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.

When you run `reorg rebuild` on a table, it locks the table for the entire time it takes to rebuild the table and its indexes. This means that you should 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 effects might be on systems that are very I/O bound.

For more information on running `reorg` commands, see the *System Administration Guide*.

Creating and maintaining indexes

Creating indexes affects performance by locking other users out of a table. The type of lock depends on the index type:

- 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

A configuration parameter configures how many buffers 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 “Configuring resources for parallel sorting” on page 218 in the *Performance and Tuning: Optimizer* book for more information.

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 perform routine index re-creations (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

If you need to re-create a clustered index or 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 that you rewrite the data rows to a complete new set of data pages. Adaptive Server can skip sorting and/or copying the table's data rows in some cases. Factors include table partitioning and on clauses used in the create index statement.

When creating an index on a nonpartitioned table, `sorted_data` and the use of any of the following clauses requires that you 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 15-1: 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 <code>sorted_data</code> only or with <code>sorted_data</code> on <code>same_segment</code>	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 <code>sorted_data</code> and <code>ignore_dup_row</code> or <code>fillfactor</code> or on <code>other_segment</code> or <code>max_rows_per_page</code>	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 reduces I/O and CPU utilization considerably.

Whenever creating a clustered index 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
- After adding rows to a table whose rows were previously deleted with `truncate table`
- After updating values in index columns

Run `update statistics` after 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 hurt performance.

See Chapter 3, “Using Statistics to Improve Performance,” in the *Performance and Tuning: Monitoring and Analyzing for Performance* book for more information.

Rebuilding indexes

Rebuilding indexes reclaims space in the B-trees. As pages are split and rows are deleted, indexes may contain many pages that contain only a few rows. Also, if your 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 re-creating the index. If the table uses data-only locking, you can run the reorg rebuild command on the table or on an individual index.

Re-create or 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 re-create a clustered index or run reorg rebuild on a data-only-locked table, all nonclustered indexes are re-created, since creating the clustered index moves rows to different pages.

You must re-create nonclustered indexes to point to the correct pages.

In many database systems, there are well-defined peak periods and off-hours. You can use off-hours to your advantage for example to:

- Delete all indexes to allow more efficient bulk inserts.
- Create a new group of indexes to help generate a set of reports.

See “Creating and maintaining indexes” on page 344 for information about configuration parameters that increase the speed of creating indexes.

Speeding index creation with *sorted_data*

If data is already sorted, you can use the *sorted_data* option for the create index command to save index creation time. You can use this option for both clustered and nonclustered indexes.

See “Creating an index on sorted data” on page 345 for more information.

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.

The following procedures can speed database creation or minimize its impact on other processes:

- Use the *for load* option to create database 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*, it copies model and then initializes all of the allocation units.

When you use *for load*, it postpones zeroing 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-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'
```

When create database copies model, it uses 2K I/O.

Note In Adaptive Server version 12.5.03 and above, the size of the large I/O buffers used by create database, alter database, load database, and dbcc checkalloc is now one allocation (256 pp), not one extent (8 pp). 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.

See the *System Administration Guide*.

Backup and recovery

All Adaptive Server backups are performed by a backup server. The backup architecture uses a client/server paradigm, with Adaptive Servers as clients to a 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* 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. It can increase the amount of I/O required.

See “Tuning the recovery interval” on page 242.

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 indexes or active triggers on the table. When you are running fast bulk copy, Adaptive Server performs reduced logging.

It does not log the actual changes to the database, only the allocation of pages. And, since there are no indexes to update, it saves all the time it would otherwise take to update indexes for each data insert and to log the changes to the index pages.

To use fast bulk copy:

- Drop any indexes; re-create them when the bulk copy completes.
- Use `alter table...disable trigger` to deactivate triggers during the copy; use `alter table...enable trigger` after the copy completes.
- Set the `select into/bulkcopy/pllsort` option with `sp_dboption`. Remember to turn the option off after the bulk copy operation completes.

During fast bulk copy, rules are not enforced, but defaults *are* enforced.

Since changes to the data are not logged, you should 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, you can use fast bulk copy to copy data into partitioned tables. For each bulk copy session, you 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 “Steps for partitioning tables” on page 117 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 are using 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

If a table has indexes or triggers, a slower version of bulk copy is automatically used. For slow bulk copy:

- You do not have to set the `select into/bulkcopy`.
- Rules are not enforced and triggers are not fired, but defaults *are* enforced.
- All data changes are logged, as well as the 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 will save the transaction dump time.

Remember that each batch is a separate transaction, so if you are not specifying a batch size, setting `trunc log on chkpt` will not help.

- Set the number of pre allocated extents configuration parameter high if you perform many large bulk copies.

See the *System Administration Guide*.

- Find the optimal network packet size.

See Chapter 3, “Networks and Performance,”.

Replacing the data in a large table

If you are replacing all the data in a large table, use the truncate table command instead of the delete command. truncate table performs reduced logging. Only the page deallocations are logged.

delete is completely logged, that is, all the changes to the data are logged.

The steps are:

- 1 Truncate the table. If the table is partitioned, you must unpartition before you can truncate it.
- 2 Drop all indexes on the table.
- 3 Load the data.
- 4 Re-create the indexes.

See “Steps for partitioning tables” on page 117 for more information on using bulk copy with partitioned tables.

Adding large amounts of data to a table

When you are adding 10 to 20 percent or more to a large table, drop the nonclustered indexes, load the data, and then re-create 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 time to perform a fast bulk copy and re-create the clustered index.

Using partitions and multiple bulk copy processes

If you are loading data into a table without indexes, you can create partitions on the table and use one bcp session for each partition.

See “Using parallel bcp to copy data into partitions” on page 110.

Impacts on other users

Bulk copying large tables in or out may affect other users' response time. If possible:

- Schedule bulk copy operations for off-hours.
- Use fast bulk copy, since it does less logging and less I/O.

Database consistency checker

It is important to run database consistency checks periodically with `dbcc`. If you back up a corrupt database, the backup is useless. But `dbcc` affects performance, since `dbcc` must acquire locks on the objects it checks.

See the *System Administration Guide* 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, a slight performance improvement may be realized by skipping this cleanup error check. To turn off error checking, enter:

```
dbcc tune(cleanup,1)
```

The final cleanup frees up any memory a task might hold. If you turn the 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 due to a spinlock contention on the "des manager" you can use the `des_bind` command to improve the scalability of the server where object descriptors are reserved for hot objects. The descriptors for these hot objects are never scavenged.

```
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 has to be in "single user mode"

When not to use this command

There are instances where this command cannot be used:

- On objects in system databases such as master and tempdb
- On system tables.

Since this bind command is not persistent, it has to be re-instantiated during startup.

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
- reorg rebuild on a table

In most cases, these commands also require space to re-create any indexes, so you need to 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

The following sections describe tools that provide information on space usage and space availability.

Overview of space requirements

Any command that copies a table's rows also re-creates all of the indexes on the table. You need space for a complete copy of the table and copies of all indexes.

These commands do not estimate how much space is needed. They stop with an error message if they run out of space on any segment used by the table or its indexes. For large tables, this could occur minutes or 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 are altering 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.

Tools for 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 room. However:

- If data modifications have created many partially-full pages, space required 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.

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 263 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 re-created 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.

The following 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 the 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 re-created.
- If `reservepagegap` values are set for indexes, these values are applied when the indexes are re-created.

Estimating the effects of space management properties

Table 15-2 shows how to estimate the effects of setting space management properties.

Table 15-2: Effects of space management properties on space use

Property	Formula	Example
fillfactor	Requires $(100/\text{fillfactor}) * \text{num_pages}$ 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 $1/\text{reservepagegap}$ 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 <code>exp_row_size</code> when converting to data-only-locking	See Table 15-3 on page 361.
exp_row_size	Increase depends on number of rows smaller than <code>exp_row_size</code> , and the average length of those rows	If <code>exp_row_size</code> 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.

For more information, see Chapter 9, “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.

The `exp_row_size` is enforced during the copy. Table 15-3 shows how the values are converted.

Table 15-3: 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: <ul style="list-style-type: none">• maximum row size• 2002/max_rows_per_page value

If there is not enough space

If there is not enough space to copy the table and re-create 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 re-create it later may require space to make a copy of the table. Re-create the nonclustered indexes after the command completes.

Tuning Asynchronous Prefetch

This chapter explains how asynchronous prefetch improves I/O performance for many types of queries by reading data and index pages into cache before they are needed by the query.

Topic	Page
How asynchronous prefetch improves performance	363
When prefetch is automatically disabled	369
Tuning Goals for asynchronous prefetch	373
Other Adaptive Server performance features	374
Special settings for asynchronous prefetch limits	377
Maintenance activities for high prefetch performance	378
Performance monitoring and asynchronous prefetch	379

How asynchronous prefetch improves performance

Asynchronous prefetch improves performance by anticipating the pages required for certain well-defined classes of database activities whose access patterns are predictable. The I/O requests for these pages are issued before the query needs them so that most pages are in cache by the time query processing needs to access the page. Asynchronous prefetch can improve performance for:

- Sequential scans, such as table scans, clustered index scans, and covered nonclustered index scans
- Access via nonclustered indexes
- Some dbcc checks and update statistics
- Recovery

Asynchronous prefetch can improve the performance of queries that access large numbers of pages, such as decision support applications, as long as the I/O subsystems on the machine are not saturated.

Asynchronous prefetch cannot help (or may help only slightly) when the I/O subsystem is already saturated or when Adaptive Server is CPU-bound. It may be used in some OLTP applications, but to a much lesser degree, since OLTP queries generally perform fewer I/O operations.

When a query in Adaptive Server needs to perform a table scan, it:

- Examines the rows on a page and the values in the rows.
- Checks the cache for the next page to be read from a table. If that page is in cache, the task continues processing. If the page is not in cache, the task issues an I/O request and sleeps until the I/O completes.
- When the I/O completes, the task moves from the sleep queue to the run queue. When the task is scheduled on an engine, Adaptive Server examines rows on the newly fetched page.

This cycle of executing and stalling for disk reads continues until the table scan completes. In a similar way, queries that use a nonclustered index process a data page, issue the I/O for the next page referenced by the index, and sleep until the I/O completes, if the page is not in cache.

This pattern of executing and then waiting for I/O slows performance for queries that issue physical I/Os for large number of pages. In addition to the waiting time for the physical I/Os to complete, the task switches on and off the engine repeatedly. This task switching adds overhead to processing.

Improving query performance by prefetching pages

Asynchronous prefetch issues I/O requests for pages before the query needs them so that most pages are in cache by the time query processing needs to access the page. If required pages are already in cache, the query does not yield the engine to wait for the physical read. (It may still yield for other reasons, but it yields less frequently.)

Based on the type of query being executed, asynchronous prefetch builds a **look-ahead set** of pages that it predicts will be needed very soon. Adaptive Server defines different look-ahead sets for each processing type where asynchronous prefetch is used.

In some cases, look-ahead sets are extremely precise; in others, some assumptions and speculation may lead to pages being fetched that are never read. When only a small percentage of unneeded pages are read into cache, the performance gains of asynchronous prefetch far outweigh the penalty for the wasted reads. If the number of unused pages becomes large, Adaptive Server detects this condition and either reduces the size of the look-ahead set or temporarily disables prefetching.

Prefetching control mechanisms in a multiuser environment

When many simultaneous queries are prefetching large numbers of pages into a buffer pool, there is a risk that the buffers fetched for one query could be flushed from the pool before they are used.

Adaptive Server tracks the buffers brought into each pool by asynchronous prefetch and the number that are used. It maintains a per-pool count of prefetched but unused buffers. By default, Adaptive Server sets an asynchronous prefetch limit of 10 percent of each pool. In addition, the limit on the number of prefetched but unused buffers is configurable on a per-pool basis.

The pool limits and usage statistics act like a governor on asynchronous prefetch to keep the cache-hit ratio high and reduce unneeded I/O. Overall, the effect is to ensure that most queries experience a high cache-hit ratio and few stalls due to disk I/O sleeps.

The following sections describe how the look-ahead set is constructed for the activities and query types that use asynchronous prefetch. In some asynchronous prefetch optimizations, allocation pages are used to build the look-ahead set.

For information on how allocation pages record information about object storage, see “Allocation pages” on page 158.

Look-ahead set during recovery

During recovery, Adaptive Server reads each log page that includes records for a transaction and then reads all the data and index pages referenced by that transaction, to verify timestamps and to roll transactions back or forward. Then, it performs the same work for the next completed transaction, until all transactions for a database have been processed. Two separate asynchronous prefetch activities speed recovery: asynchronous prefetch on the log pages themselves and asynchronous prefetch on the referenced data and index pages.

Prefetching log pages

The transaction log is stored sequentially on disk, filling extents in each allocation unit. Each time the recovery process reads a log page from a new allocation unit, it prefetches all the pages on that allocation unit that are in use by the log.

In databases that do not have a separate log segment, log and data extents may be mixed on the same allocation unit. Asynchronous prefetch still fetches all the log pages on the allocation unit, but the look-ahead sets may be smaller.

Prefetching data and index pages

For each transaction, Adaptive Server scans the log, building the look-ahead set from each referenced data and index page. While one transaction's log records are being processed, asynchronous prefetch issues requests for the data and index pages referenced by subsequent transactions in the log, reading the pages for transactions ahead of the current transaction.

Note Recovery uses only the pool in the default data cache. See “Setting limits for recovery” on page 377 for more information.

Look-ahead set during sequential scans

Sequential scans include table scans, clustered index scans, and covered nonclustered index scans.

During table scans and clustered index scans, asynchronous prefetch uses allocation page information about the pages used by the object to construct the look-ahead set. Each time a page is fetched from a new allocation unit, the look-ahead set is built from all the pages on that allocation unit that are used by the object.

The number of times a sequential scan hops between allocation units is kept to measure fragmentation of the page chain. This value is used to adapt the size of the look-ahead set so that large numbers of pages are prefetched when fragmentation is low, and smaller numbers of pages are fetched when fragmentation is high. For more information, see “Page chain fragmentation” on page 371.

Look-ahead set during nonclustered index access

When using a nonclustered index to access rows, asynchronous prefetch finds the page numbers for all qualified index values on a nonclustered index leaf page. It builds the look-ahead set from the unique list of all the pages that are needed.

Asynchronous prefetch is used only if two or more rows qualify.

If a nonclustered index access requires several leaf-level pages, asynchronous prefetch requests are also issued on the leaf pages.

Look-ahead set during *dbcc* checks

Asynchronous prefetch is used during the following *dbcc* checks:

- *dbcc checkalloc*, which checks allocation for all tables and indexes in a database, and the corresponding object-level commands, *dbcc tablealloc* and *dbcc indexalloc*
- *dbcc checkdb*, which checks all tables and index links in a database, and *dbcc checktable*, which checks individual tables and their indexes

Allocation checking

The dbcc commands checkalloc, tablealloc and indexalloc, which check page allocations validate information on the allocation page. The look-ahead set for the dbcc operations that check allocation is similar to the look-ahead set for other sequential scans. When the scan enters a different allocation unit for the object, the look-ahead set is built from all the pages on the allocation unit that are used by the object.

checkdb and checktable

The dbcc checkdb and dbcc checktable commands check the page chains for a table, building the look-ahead set in the same way as other sequential scans.

If the table being checked has nonclustered indexes, they are scanned recursively, starting at the root page and following all pointers to the data pages. When checking the pointers from the leaf pages to the data pages, the dbcc commands use asynchronous prefetch in a way that is similar to nonclustered index scans. When a leaf-level index page is accessed, the look-ahead set is built from the page IDs of all the pages referenced on the leaf-level index page.

Look-ahead set minimum and maximum sizes

The size of a look-ahead set for a query at a given point in time is determined by several factors:

- The type of query, such as a sequential scan or a nonclustered index scan
- The size of the pools used by the objects that are referenced by the query and the prefetch limit set on each pool
- The fragmentation of tables or indexes, in the case of operations that perform scans
- The recent success rate of asynchronous prefetch requests and overload conditions on I/O queues and server I/O limits

Table 16-1 summarizes the minimum and maximum sizes for different type of asynchronous prefetch usage.

Table 16-1: Look-ahead set sizes

Access type	Action	Look-ahead set sizes
Table scan Clustered index scan Covered leaf level scan	Reading a page from a new allocation unit	Minimum is 8 pages needed by the query Maximum is the smaller of: <ul style="list-style-type: none"> • The number of pages on an allocation unit that belong to an object. • The pool prefetch limits
Nonclustered index scan	Locating qualified rows on the leaf page and preparing to access data pages	Minimum is 2 qualified rows Maximum is the smaller of: <ul style="list-style-type: none"> • The number of unique page numbers on qualified rows on the leaf index page • The pool's prefetch limit
Recovery	Recovering a transaction	Maximum is the smaller of: <ul style="list-style-type: none"> • All of the data and index pages touched by a transaction undergoing recovery • The prefetch limit of the pool in the default data cache
	Scanning the transaction log	Maximum is all pages on an allocation unit belonging to the log
dbcc tablealloc, indexalloc, and checkalloc	Scanning the page chain	Same as table scan
dbcc checktable and checkdb	Scanning the page chain	Same as table scan
	Checking nonclustered index links to data pages	All of the data pages referenced on a leaf level page.

When prefetch is automatically disabled

Asynchronous prefetch attempts to fetch needed pages into buffer pools without flooding the pools or the I/O subsystem and without reading unneeded pages. If Adaptive Server detects that prefetched pages are being read into cache but not used, it temporarily limits or discontinues asynchronous prefetch.

Flooding pools

For each pool in the data caches, a configurable percentage of buffers can be read in by asynchronous prefetch and held until their first use. For example, if a 2K pool has 4000 buffers, and the limit for the pool is 10 percent, then, at most, 400 buffers can be read in by asynchronous prefetch and remain unused in the pool. If the number of nonaccessed prefetched buffers in the pool reaches 400, Adaptive Server temporarily discontinues asynchronous prefetch for that pool.

As the pages in the pool are accessed by queries, the count of unused buffers in the pool drops, and asynchronous prefetch resumes operation. If the number of available buffers is smaller than the number of buffers in the look-ahead set, only that many asynchronous prefetches are issued. For example, if 350 unused buffers are in a pool that allows 400, and a query's look-ahead set is 100 pages, only the first 50 asynchronous prefetches are issued.

This keeps multiple asynchronous prefetch requests from flooding the pool with requests that flush pages out of cache before they can be read. The number of asynchronous I/Os that cannot be issued due to the per-pool limits is reported by `sp_sysmon`.

I/O system overloads

Adaptive Server and the operating system place limits on the number of outstanding I/Os for the server as a whole and for each engine. The configuration parameters `max async i/os per server` and `max async i/os per engine` control these limits for Adaptive Server. See your operating system documentation for more information on configuring them for your hardware.

The configuration parameter `disk i/o structures` controls the number of disk control blocks that Adaptive Server reserves. Each physical I/O (each buffer read or written) requires one control block while it is in the I/O queue.

See the *System Administration Guide*.

If Adaptive Server tries to issue asynchronous prefetch requests that would exceed max async i/os per server, max async i/os per engine, or disk i/o structures, it issues enough requests to reach the limit and discards the remaining requests. For example, if only 50 disk I/O structures are available, and the server attempts to prefetch 80 pages, 50 requests are issued, and the other 30 are discarded.

sp_sysmon reports the number of times these limits are exceeded by asynchronous prefetch requests. See “Asynchronous prefetch activity report” on page 86 in the book *Performance and Tuning: Monitoring and Analyzing for Performance*.

Unnecessary reads

Asynchronous prefetch tries to avoid unnecessary physical reads. During recovery and during nonclustered index scans, look-ahead sets are exact, fetching only the pages referenced by page number in the transaction log or on index pages.

Look-ahead sets for table scans, clustered index scans, and dbcc checks are more speculative and may lead to unnecessary reads. During sequential scans, unnecessary I/O can take place due to:

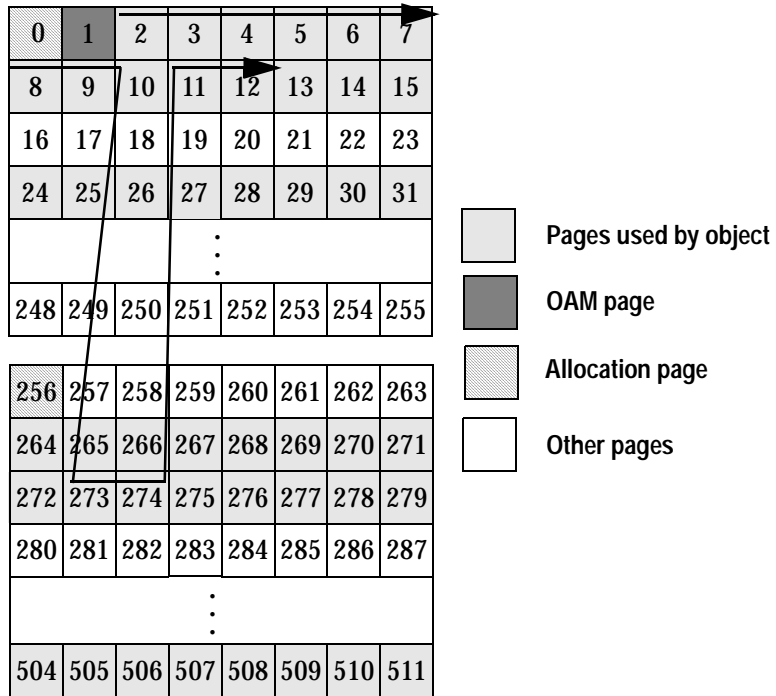
- Page chain fragmentation on allpages-locked tables
- Heavy cache utilization by multiple users

Page chain fragmentation

Adaptive Server’s page allocation mechanism strives to keep pages that belong to the same object close to each other in physical storage by allocating new pages on an extent already allocated to the object and by allocating new extents on allocation units already used by the object.

However, as pages are allocated and deallocated, page chains on data-only-locked tables can develop kinks. Figure 16-1 shows an example of a kinked page chain between extents in two allocation units.

Figure 16-1: A kink in a page chain crossing allocation units



In Figure 16-1, when a scan first needs to access a page from allocation unit 0, it checks the allocation page and issues asynchronous I/Os for all the pages used by the object it is scanning, up to the limit set on the pool. As the pages become available in cache, the query processes them in order by following the page chain. When the scan reaches page 10, the next page in the page chain, page 273, belongs to allocation unit 256.

When page 273 is needed, allocation page 256 is checked, and asynchronous prefetch requests are issued for all the pages in that allocation unit that belong to the object.

When the page chain points back to a page in allocation unit 0, there are two possibilities:

- The prefetched pages from allocation unit 0 are still in cache, and the query continues processing with no unneeded physical I/Os.

- The prefetch pages from allocation unit 0 have been flushed from the cache by the reads from allocation unit 256 and other I/Os taking place by other queries that use the pool. The query must reissue the prefetch requests. This condition is detected in two ways:
 - Adaptive Server's count of the hops between allocation pages now equals two. It uses the ratio between the count of hops and the prefetched pages to reduce the size of the look-ahead set, so fewer I/Os are issued.
 - The count of prefetched but unused pages in the pool is likely to be high, so asynchronous prefetch may be temporarily discontinued or reduced, based on the pool's limit.

Tuning Goals for asynchronous prefetch

Choosing optimal pool sizes and prefetch percentages for buffer pools can be key to achieving improved performance with asynchronous prefetch. When multiple applications are running concurrently, a well-tuned prefetching system balances pool sizes and prefetch limits to accomplish these goals:

- Improved system throughput
- Better performance by applications that use asynchronous prefetch
- No performance degradation in applications that do not use asynchronous prefetch

Configuration changes to pool sizes and the prefetch limits for pools are dynamic, allowing you to make changes to meet the needs of varying workloads. For example, you can configure asynchronous prefetch for good performance during recovery or dbcc checking and reconfigure afterward without needing to restart Adaptive Server.

See “Setting limits for recovery” on page 377 and “Setting limits for dbcc” on page 378 for more information.

Commands for configuration

Asynchronous prefetch limits are configured as a percentage of the pool in which prefetched but unused pages can be stored. There are two configuration levels:

- The server-wide default, set with the configuration parameter `global async prefetch limit`. When you first install, the default value for `global async prefetch limit` is 10 (percent).

For more information, see of the *System Administration Guide*.

- A per-pool override, set with `sp_poolconfig`. To see the limits set for each pool, use `sp_cacheconfig`.

For more information, see of the *System Administration Guide*.

Changing asynchronous prefetch limits takes effect immediately, and does not require a reboot. Both the global and per-pool limits can also be configured in the configuration file.

Other Adaptive Server performance features

This section covers the interaction of asynchronous prefetch with other Adaptive Server performance features.

Large I/O

The combination of large I/O and asynchronous prefetch can provide rapid query processing with low I/O overhead for queries performing table scans and for `dbcc` operations.

When large I/O prefetches all the pages on an allocation unit, the minimum number of I/Os for the entire allocation unit is:

- 31 16K I/Os

- 7 2K I/Os, for the pages that share an extent with the allocation page

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

Sizing and limits for the 16k pool

Performing 31 16K prefetches with the default asynchronous prefetch limit of 10 percent of the buffers in the pool requires a pool with at least 310 16K buffers. If the pool is smaller, or if the limit is lower, some prefetch requests will be denied. To allow more asynchronous prefetch activity in the pool, you can configure a larger pool or a larger prefetch limit for the pool.

If multiple overlapping queries perform table scans using the same pool, the number of unused, prefetched pages allowed in the pool needs to be higher. The queries are probably issuing prefetch requests at slightly staggered times and are at different stages in reading the accessed pages. For example, one query may have just prefetched 31 pages, and have 31 unused pages in the pool, while an earlier query has only 2 or 3 unused pages left. To start your tuning efforts for these queries, assume one-half the number of pages for a prefetch request multiplied by the number of active queries in the pool.

Limits for the 2K pool

Queries using large I/O during sequential scans may still need to perform 2K I/O:

- When a scan enters a new allocation unit, it performs 2K I/O on the 7 pages in the unit that share space with the allocation page.
- If pages from the allocation unit already reside in the 2K pool when the prefetch requests are issued, the pages that share that extent must be read into the 2K pool.

If the 2K pool has its asynchronous prefetch limit set to 0, the first 7 reads are performed by normal asynchronous I/O, and the query sleeps on each read if the pages are not in cache. Set the limits on the 2K pool high enough that it does not slow prefetching performance.

Fetch-and-discard (MRU) scans

When a scan uses MRU replacement policy, buffers are handled in a special manner when they are read into the cache by asynchronous prefetch. First, pages are linked at the MRU end of the chain, rather than at the wash marker. When the query accesses the page, the buffers are re linked into the pool at the wash marker. This strategy helps to avoid cases where heavy use of a cache flushes prefetched buffers linked at the wash marker before they can be used. It has little impact on performance, unless large numbers of unneeded pages are being prefetched. In this case, the prefetched pages are more likely to flush other pages from cache.

Parallel scans and large I/Os

The demand on buffer pools can become higher with parallel queries. With serial queries operating on the same pools, it is safe to assume that queries are issued at slightly different times and that the queries are in different stages of execution: some are accessing pages already in cache, and others are waiting on I/O.

Parallel execution places different demands on buffer pools, depending on the type of scan and the degree of parallelism. Some parallel queries are likely to issue a large number of prefetch requests simultaneously.

Hash-based table scans

Hash-based table scans on allpages-locked tables have multiple worker processes accessing the same page chain. Each worker process checks the page ID of each page in the table, but examines only the rows on those pages where page ID matches the hash value for the worker process.

The first worker process that needs a page from a new allocation unit issues a prefetch request for all pages from that unit. When the scans of other worker processes also need pages from that allocation unit, they will either find that the pages they need are already in I/O or already in cache. As the first scan to complete enters the next unit, the process is repeated.

As long as one worker process in the family performing a hash-based scan does not become stalled (waiting for a lock, for example), the hash-based table scans do not place higher demands on the pools than they place on serial processes. Since the multiple processes may read the pages much more quickly than a serial process does, they change the status of the pages from unused to used more quickly.

Partition-based scans

Partition-based scans are more likely to create additional demands on pools, since multiple worker processes may be performing asynchronous prefetching on different allocation units. On partitioned tables on multiple devices, the per-server and per-engine I/O limits are less likely to be reached, but the per-pool limits are more likely to limit prefetching.

Once a parallel query is parsed and compiled, it launches its worker processes. If a table with 4 partitions is being scanned by 4 worker processes, each worker process attempts to prefetch all the pages in its first allocation unit. For the performance of this single query, the most desirable outcome is that the size and limits on the 16K pool are sufficiently large to allow 124 (31*4) asynchronous prefetch requests, so all of the requests succeed. Each of the worker processes scans the pages in cache quickly, moving onto new allocation units and issuing more prefetch requests for large numbers of pages.

Special settings for asynchronous prefetch limits

You may want to change asynchronous prefetch configuration temporarily for specific purposes, including:

- Recovery
- dbcc operations that use asynchronous prefetch

Setting limits for recovery

During recovery, Adaptive Server uses only the 2K pool of the default data cache. If you shut down the server using `shutdown with nowait`, or if the server goes down due to power failure or machine failure, the number of log records to be recovered may be quite large.

To speed recovery, you can edit the configuration file to do one or both of the following:

- Increase the size of the 2K pool in the default data cache by reducing the size of other pools in the cache
- Increase the prefetch limit for the 2K pool

Both of these configuration changes are dynamic, so you can use `sp_poolconfig` to restore the original values after recovery completes, without restarting Adaptive Server. The recovery process allows users to log into the server as soon as recovery of the master database is complete. Databases are recovered one at a time and users can begin using a particular database as soon as it is recovered. There may be some contention if recovery is still taking place on some databases, and user activity in the 2K pool of the default data cache is heavy.

Setting limits for *dbcc*

If you are performing database consistency checking at a time when other activity on the server is low, configuring high asynchronous prefetch limits on the pools used by `dbcc` can speed consistency checking.

`dbcc checkalloc` can use special internal 16K buffers if there is no 16K pool in the cache for the appropriate database. If you have a 2K pool for a database, and no 16K pool, set the local prefetch limit to 0 for the pool while executing `dbcc checkalloc`. Use of the 2K pool instead of the 16K internal buffers may actually hurt performance.

Maintenance activities for high prefetch performance

Page chains for all pages-locked tables and the leaf levels of indexes develop kinks as data modifications take place on the table. In general, newly created tables have few kinks. Tables where updates, deletes, and inserts that have caused page splits, new page allocations, and page deallocations are likely to have cross-allocation unit page chain kinks. If more than 10 to 20 percent of the original rows in a table have been modified, you should determine if kinked page chains are reducing asynchronous prefetch effectiveness. If you suspect that page chain kinks are reducing asynchronous prefetch performance, you may need to re-create indexes or reload tables to reduce kinks.

Eliminating kinks in heap tables

For allpages-locked heaps, page allocation is generally sequential, unless pages are deallocated by deletes that remove all rows from a page. These pages may be reused when additional space is allocated to the object. You can create a clustered index (and drop it, if you want the table stored as a heap) or bulk copy the data out, truncate the table, and copy the data in again. Both activities compress the space used by the table and eliminate page-chain kinks.

Eliminating kinks in clustered index tables

For clustered indexes, page splits and page deallocations can cause page chain kinks. Rebuilding clustered indexes does not necessarily eliminate all cross-allocation page linkages. Use `fillfactor` for clustered indexes where you expect growth, to reduce the number of kinks resulting from data modifications.

Eliminating kinks in nonclustered indexes

If your query mix uses covered index scans, dropping and re-creating nonclustered indexes can improve asynchronous prefetch performance, once the leaf-level page chain becomes fragmented.

Performance monitoring and asynchronous prefetch

The output of `statistics io` reports the number physical reads performed by asynchronous prefetch and the number of reads performed by normal asynchronous I/O. In addition, `statistics io` reports the number of times that a search for a page in cache was found by the asynchronous prefetch without holding the cache spinlock.

See “Reporting physical and logical I/O statistics” on page 63 in the *Performance and Tuning: Monitoring and Analyzing for Performance* book for more information.

sp_sysmon report contains information on asynchronous prefetch in both the “Data Cache Management” section and the “Disk I/O Management” section.

If you are using sp_sysmon to evaluate asynchronous prefetch performance, you may see improvements in other performance areas, such as:

- Much higher cache hit ratios in the pools where asynchronous prefetch is effective
- A corresponding reduction in context switches due to cache misses, with voluntary yields increasing
- A possible reduction in lock contention. Tasks keep pages locked during the time it takes to perform I/O for the next page needed by the query. If this time is reduced because asynchronous prefetch increases cache hits, locks will be held for a shorter time.

See “Data cache management” on page 82 and “Disk I/O management” on page 102 in the *Performance and Tuning: Monitoring and Analyzing for Performance* book for more information.

This chapter discusses the performance issues associated with using the tempdb database. tempdb is used by Adaptive Server users. Anyone can create objects in tempdb. Many processes use it silently. It is a server-wide resource that is used primarily for internal sorts processing, creating worktables, reformatting, and for storing temporary tables and indexes created by users.

Many applications use stored procedures that create tables in tempdb to expedite complex joins or to perform other complex data analysis that is not easily performed in a single step.

Topic	Page
How management of tempdb affects performance	381
Types and uses of temporary tables	382
Initial allocation of tempdb	384
Sizing the tempdb	385
Placing tempdb	386
Dropping the master device from tempdb segments	386
Binding tempdb to its own cache	387
Temporary tables and locking	388
Minimizing logging in tempdb	389
Optimizing temporary tables	390

How management of *tempdb* affects performance

Good management of tempdb is critical to the overall performance of Adaptive Server. tempdb cannot be overlooked or left in a default state. It is the most dynamic database on many servers and should receive special attention.

If planned for in advance, most problems related to tempdb can be avoided. These are the kinds of things that can go wrong if tempdb is not sized or placed properly:

- tempdb fills up frequently, generating error messages to users, who must then resubmit their queries when space becomes available.
- Sorting is slow, and users do not understand why their queries have such uneven performance.
- User queries are temporarily locked from creating temporary tables because of locks on system tables.
- Heavy use of tempdb objects flushes other pages out of the data cache.

Main solution areas for *tempdb* performance

These main areas can be addressed easily:

- Sizing tempdb correctly for all Adaptive Server activity
- Placing tempdb optimally to minimize contention
- Binding tempdb to its own data cache
- Minimizing the locking of resources within tempdb

Types and uses of temporary tables

The use or misuse of user-defined temporary tables can greatly affect the overall performance of Adaptive Server and your applications.

Temporary tables can be quite useful, often reducing the work the server has to do. However, temporary tables can add to the size requirement of tempdb. Some temporary tables are truly temporary, and others are permanent.

tempdb is used for three types of tables:

- Truly temporary tables
- Regular user tables
- Worktables

Truly temporary tables

You can create truly temporary tables by using “#” as the first character of the table name:

```
create table #temptable (...)
```

or:

```
select select_list
into #temptable ...
```

Temporary tables:

- Exist only for the duration of the user session or for the scope of the procedure that creates them
- Cannot be shared between user connections
- Are automatically dropped at the end of the session or procedure (or can be dropped manually)

When you create indexes on temporary tables, the indexes are stored in tempdb:

```
create index tempix on #temptable(col1)
```

Regular user tables

You can create regular user tables in tempdb by specifying the database name in the command that creates the table:

```
create table tempdb..temptable (...)
```

or:

```
select select_list
into tempdb..temptable
```

Regular user tables in tempdb:

- Can persist across sessions
- Can be used by bulk copy operations
- Can be shared by granting permissions on them
- Must be explicitly dropped by the owner (otherwise, they are removed when Adaptive Server is restarted)

You can create indexes in tempdb on permanent temporary tables:

```
create index tempix on tempdb..temptable(col1)
```

Worktables

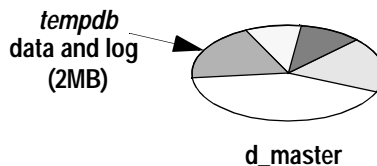
Worktables are automatically created in tempdb by Adaptive Server for merge joins, sorts, and other internal server processes. These tables:

- Are never shared
- Disappear as soon as the command completes

Initial allocation of *tempdb*

When you install Adaptive Server, tempdb is 2MB, and is located completely on the master device, as shown in Figure 17-1. This 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. It can be altered onto the master device or other devices. Depending on your needs, you may want to stripe tempdb across several devices.

Figure 17-1: 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

```

Sizing the *tempdb*

tempdb needs to be big enough 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
- Temporary tables (those created with “#” as the first character of their names)
- Indexes on temporary tables
- Regular user tables in *tempdb*
- 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 *tempdb* to:

- Denormalize several tables into a few temporary tables
- Normalize a denormalized table to do aggregate processing

For most applications, make *tempdb* 20 to 25% of the size of your user databases to provide enough space for these uses.

Placing *tempdb*

Keep tempdb on separate physical disks from your critical application databases. Use the fastest disks available. If your platform supports solid state devices and your 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 your tempdb database is heavily used.

Dropping the master device from *tempdb* segments

By default, the system, default, and logsegment segments for tempdb include its 2MB allocation on the master device. When you allocate new devices to tempdb, they automatically become part of all three segments. Once you allocate a second device to tempdb, you can drop the master device from the default 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 that the default segment no longer includes the master device, issue this command:

```
select dbid, name, segmap
```



```

from sysusages, sysdevices
where sysdevices.low <= sysusages.size + vstart
  and sysdevices.high >= sysusages.size + vstart -1
  and dbid = 2
  and (status = 2 or status = 3)

```

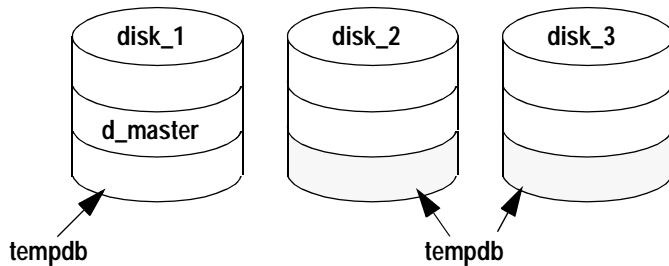
The `segmap` column should report “1” for any allocations on the master device, indicating that only the system segment still uses the device:

dbid	name	segmap
2	master	1
2	tune3	7

Using multiple disks for parallel query performance

If `tempdb` spans multiple devices, as shown in Figure 17-2, you can take advantage of parallel query performance for some temporary tables or worktables.

Figure 17-2: *tempdb* spanning disks



Binding *tempdb* to its own cache

Under normal Adaptive Server use, `tempdb` makes heavy use of the data cache as temporary tables are created, populated, and then dropped.

Assigning `tempdb` 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

See “Examining cache needs for tempdb” on page 232 for more information.

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 the *System Administration Guide*.

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
```

Temporary tables and locking

Creating or dropping temporary tables and their indexes can cause lock contention on the system tables in tempdb. When users create tables in tempdb, information about the tables must be stored in system tables such as sysobjects, syscolumns, and sysindexes. If multiple user processes are creating and dropping tables in tempdb, heavy contention can occur on the system tables. Worktables created internally do not store information in system tables.

If contention for tempdb system tables is a problem with applications that must repeatedly create and drop the same set of temporary tables, try creating the tables at the start of the application. Then use `insert...select` to populate them, and `truncate table` to remove all the data rows. Although `insert...select` requires logging and is slower than `select into`, it can provide a solution to the locking problem.

Minimizing logging in *tempdb*

Even though the trunc log on checkpoint database option is turned on in *tempdb*, changes to *tempdb* are still written to the transaction log. You can reduce log activity in *tempdb* by:

- Using `select into` instead of `create table and insert`
- Selecting only the columns you need into the temporary tables

With *select into*

When you create and populate temporary tables in *tempdb*, 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 *tempdb* 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.

By using shorter rows

If the application creating tables in *tempdb* uses only a few columns of a table, you can minimize the number and size of log records by:

- Selecting just 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

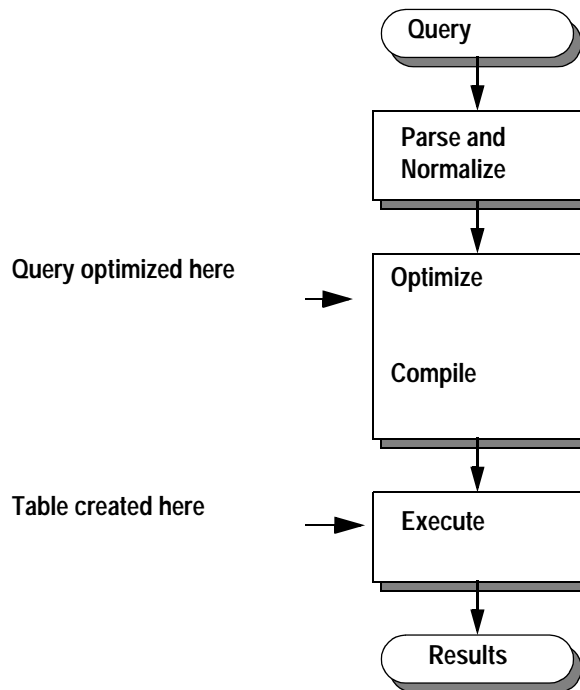
Both of these suggestions also keep the size of the tables themselves smaller.

Optimizing temporary tables

Many uses of temporary tables are simple and brief and require little optimization. But if your applications require multiple accesses to tables in tempdb, you should 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, the table has not yet been created when the query is optimized, as shown in Figure 17-3. This applies to both temporary tables and regular user tables.

Figure 17-3: 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 `tempdb`. 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 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. Here are the steps:

- 1 Create the temporary table outside the procedure. It can be empty; it just needs to exist and to 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.

Index

Symbols

- # (pound sign)
 - temporary table identifier prefix 383

Numerics

- 4K memory pool, transaction log and 234

A

- access
 - index 152
 - memory and disk speeds 205
 - optimizer methods 151
- Adaptive Server
 - column size 12, 135
 - logical page sizes 12, 134, 153, 154
 - number of groups 13, 135
 - number of logins 13, 135
 - number of users 13, 135
- affinity
 - CPU 44, 56
 - engine example 76
- aggregate functions
 - denormalization and performance 144
 - denormalization and temporary tables 385
- aging
 - data cache 216
 - procedure cache 212
- algorithm 59
 - guidelines 62
- allocating memory 210
- allocation
 - dynamic allocation 209
- allocation map. *See* Object Allocation Map (OAM)
 - pages
- allocation pages 158
- allocation units 156, 158
 - database creation and 348
- ALS
 - log writer 50, 324
 - user log cache 48, 322
 - when to use 48, 322
- ALS, *see* Asynchronous Log Service 47, 320
- alter table** command
 - lock** option and **fillfactor** and 188
 - partition** clause 106
 - reservepagegap** for indexes 197
 - unpartition** 107
- APL tables. *See* all pages locking
- application design
 - cursors and 342
 - denormalization for 143
 - DSS and OLTP 221
 - managing denormalized data with 149
 - network packet size and 29
 - primary keys and 310
 - procedure cache sizing 213
 - SMP servers 57
 - temporary tables in 385
- application execution precedence 67, 85–87
 - environment analysis 65
 - scheduling and 75
 - system procedures 71
- application queues. *See* application execution precedence
- architecture
 - multithreaded 35
- artificial columns 319
- assigning execution precedence 67
- asynchronous prefetch 363, 374
 - dbcc** and 367, 378
 - during recovery 366
 - fragmentation and 371
 - hash-based scans and 376
 - large I/O and 374
 - look-ahead set 364

Index

- maintenance for 378
 - MRU replacement strategy and 376
 - nonclustered indexes and 367
 - page chain fragmentation and 371
 - page chain kinks and 371, 378
 - parallel query processing and 376
 - partition-based scans and 377
 - performance monitoring 380
 - pool limits and 370
 - recovery and 377
 - sequential scans and 366
 - tuning goals 373
 - @@pack_received global variable 29
 - @@pack_sent global variable 29
 - @@packet_errors global variable 29
 - attributes
 - execution classes 69
 - auditing
 - disk contention and 91
 - performance effects 243
 - queue, size of 245
- ## B
- Backup Server 350
 - backups
 - network activity from 31
 - planning 7
 - base priority 69
 - batch processing
 - bulk copy and 353
 - managing denormalized data with 150
 - temporary tables and 391
 - bcp** (bulk copy utility) 352
 - heap tables and 169
 - large I/O for 227
 - parallel 110
 - partitioned tables and 110
 - reclaiming space with 181
 - temporary tables 383
 - binary expressions xxii
 - binding
 - caches 220, 239
 - objects to data caches 174
 - tempdb* 221, 388
 - transaction logs 221
 - B-trees, index
 - nonclustered indexes 285
 - buffers
 - allocation and caching 177
 - chain of 174
 - procedure (“proc”) 213
 - bulk copying. *See bcp* (bulk copy utility)
 - business models and logical database design 133
- ## C
- cache hit ratio
 - cache replacement policy and 231
 - data cache 218
 - procedure cache 213
 - cache replacement policy 229
 - defined 229
 - indexes 230
 - lookup tables 230
 - transaction logs 230
 - cache replacement strategy 174–179, 229
 - cache, procedure
 - cache hit ratio 213
 - errors 213
 - query plans in 212
 - size report 212
 - sizing 213
 - caches, data 215–241
 - aging in 174
 - binding objects to 174
 - cache hit ratio 218
 - data modification and 177, 217
 - deletes on heaps and 178
 - guidelines for named 230
 - hot spots bound to 220
 - I/O configuration 173, 227
 - inserts to heaps and 177
 - joins and 176
 - large I/O and 225
 - MRU replacement strategy 175
 - named 220–240
 - page aging in 215
 - pools in 173, 227
 - spinlocks on 221

- strategies chosen by optimizer 228
- tempdb* bound to own 221, 388
- transaction log bound to own 221
- updates to heaps and 178
- wash marker 174
- chain of buffers (data cache) 174
- chains of pages
 - overflow pages and 282
 - placement 90
 - unpartitioning 107
- character expressions xxii
- checkpoint process 216
 - housekeeper task and 51
- client
 - connections 35
 - packet size specification 29
 - task 36
- client/server architecture 21
- close** command
 - memory and 330
- close on endtran** option, **set** 342
- cluster ratio
 - reservepagegap** and 194, 199
- clustered indexes 274
 - asynchronous prefetch and scans 366
 - computing number of data pages 264
 - computing number of pages 258
 - computing size of rows 259
 - delete operations 283
 - estimating size of 257, 263
 - exp_row_size** and row forwarding 189–194
 - fillfactor effect on 268
 - guidelines for choosing 306
 - insert operations and 278
 - order of key values 277
 - overflow pages and 282
 - overhead 180
 - page reads 278
 - partitioned tables and 108
 - performance and 180
 - reclaiming space with 181
 - reducing forwarded rows 189–194
 - scans and asynchronous prefetch 366
 - segments and 98
 - select operations and 277
 - size of 251, 260
 - structure of 276
- collapsing tables 145
- column size 12, 135
- columns
 - artificial 319
 - datatype sizes and 258, 264
 - derived 144
 - fixed- and variable-length 258
 - fixed-length 264
 - redundant in database design 144
 - splitting tables 148
 - unindexed 153
 - values in, and normalization 137
 - variable-length 264
- commands for configuration 374
- compiled objects 213
 - data cache size and 214
- composite indexes 312
 - advantages of 314
- concurrency
 - SMP environment 57
- configuration (Server)
 - memory 206
- configuration (server)
 - housekeeper task 51
 - I/O 225
 - named data caches 220
 - network packet size 27
 - number of rows per page 204
- connections
 - client 35
 - cursors and 342
 - packet size 27
- consistency
 - data and performance 150
- constants xxii
- constraints
 - primary key** 304
 - unique 304
- contention
 - avoiding with clustered indexes 273
 - data cache 232
 - disk I/O 93, 242
 - I/O device 93
 - logical devices and 90
 - max_rows_per_page** and 203

Index

- partitions to avoid 99
- SMP servers and 57
- spinlock 232
- system tables in *tempdb* 388
- transaction log writes 182
- underlying problems 91
- control pages for partitioned tables
 - updating statistics on 116
- controller, device 93
- conventions
 - used in manuals xix
- covered queries
 - index covering 152
- covering nonclustered indexes
 - asynchronous prefetch and 366
 - configuring I/O size for 237
 - rebuilding 347
- CPU
 - affinity 56
- cpu grace time** configuration parameter
 - CPU yields and 43
- CPU usage
 - housekeeper task and 50
 - monitoring 53
 - sp_monitor** system procedure 53
- cpuaffinity (dbcc tune parameter)** 56
- create clustered index** command
 - sorted_data** and **fillfactor** interaction 188
 - sorted_data** and **reservepagegap** interaction 200–202
- create database** command
 - parallel I/O 90
- create index** command
 - distributing data with 108
 - fillfactor** and 183–188
 - locks acquired by 344
 - parallel configuration and 108
 - parallel sort and 108
 - reservepagegap** option 197
 - segments and 345
 - sorted_data** option 345
- create table** command
 - exp_row_size** option 190
 - reservepagegap** option 196
 - space management properties 190
- cursor rows** option, **set** 341

- cursors
 - execute 330
 - Halloween problem 332
 - indexes and 331
 - isolation levels and 338
 - locking and 328
 - modes 331
 - multiple 342
 - read-only 331
 - stored procedures and 330
 - updatable 331

D

- data
 - consistency 150
 - little-used 147
 - max_rows_per_page** and storage 203
 - storage 93, 151–182
 - uniqueness 273
- data caches 215–241
 - aging in 174
 - binding objects to 174
 - cache hit ratio 218
 - data modification and 177, 217
 - deletes on heaps and 178
 - fetch-and-discard strategy 175
 - guidelines for named 230
 - hot spots bound to 220
 - inserts to heaps and 177
 - joins and 176
 - large I/O and 225
 - named 220–240
 - page aging in 215
 - sizing 222–238
 - spinlocks on 221
 - strategies chosen by optimizer 228
 - tempdb* bound to own 221, 387, 388
 - transaction log bound to own 221
 - updates to heaps and 178
 - wash marker 174
- data integrity
 - application logic for 149
 - denormalization effect on 142
 - managing 148

- data modification
 - data caches and 177, 217
 - heap tables and 168
 - log space and 351
 - nonclustered indexes and 311
 - number of indexes and 299
 - recovery interval and 242
 - transaction log and 181
- data pages 153–181
 - clustered indexes and 276
 - computing number of 258, 264
 - fillfactor effect on 268
 - full, and insert operations 279
 - limiting number of rows on 203
 - linking 167
 - partially full 180
 - text and image 156
- database design 133–150
 - collapsing tables 145
 - column redundancy 144
 - indexing based on 317
 - logical keys and index keys 306
 - normalization 135
- database devices 92
 - parallel queries and 93
 - sybsecurity* 94
 - tempdb* 94
- database objects
 - binding to caches 174
 - placement 89–132
 - placement on segments 89
 - storage 151–182
- databases
 - See also* database design
 - creation speed 348
 - devices and 93
 - placement 89
- datatypes
 - choosing 310, 319
 - numeric compared to character 319
- dbcc** (database consistency checker)
 - configuring asynchronous prefetch for 378
- dbcc** (database consistency checker)
 - asynchronous prefetch and 367
 - large I/O for 227
- dbcc (engine)** command 55
- dbcc tune**
 - cleanup** 355
 - cpuaffinity** 56
 - des_bind** 356
- deallocate cursor** command
 - memory and 330
- decision support system (DSS) applications
 - execution preference 86
 - named data caches for 221
 - network packet size for 27
- declare cursor** command
 - memory and 330
- default exp_row_size percent** configuration parameter 191
- default fill factor percentage** configuration parameter 186
- default settings
 - audit queue size 245
 - auditing 244
 - max_rows_per_page** 203
 - network packet size 27
- delete operations
 - clustered indexes 283
 - heap tables 170
 - nonclustered indexes 290
 - object size and 249
- denormalization 141
 - application design and 149
 - batch reconciliation and 150
 - derived columns 144
 - disadvantages of 143
 - duplicating tables and 146
 - management after 148
 - performance benefits of 143
 - processing costs and 142
 - redundant columns 144
 - techniques for 144
 - temporary tables and 385
- derived columns 144
- devices
 - adding for partitioned tables 123, 128
 - object placement on 89
 - partitioned tables and 128
 - RAID 103
 - throughput, measuring 103
 - using separate 58

Index

- dirty pages
 - checkpoint process and 216
 - wash area and 215
 - disk devices
 - performance and 89–132
 - disk I/O
 - performing 46
 - disk i/o structures** configuration parameter
 - asynchronous prefetch and 370
 - disk mirroring
 - device placement 95
 - performance and 90
 - DSS applications
 - See Decision Support Systems
 - duplication
 - tables 146
 - dynamic memory allocation 209
- ## E
- EC
 - attributes 69
 - engine affinity, task 69, 71
 - example 72
 - engine resources
 - results analysis and tuning 66
 - engine resources, distribution 59
 - engines 36
 - CPU affinity 56
 - defined 36
 - functions and scheduling 44
 - network 45
 - scheduling 44
 - taking offline 55
 - environment analysis 65
 - I/O-intensive and CPU-intensive execution objects 64
 - intrusive and unintrusive 64
 - environment analysis and planning 63
 - error logs
 - procedure cache size in 212
 - error messages
 - procedure cache 213
 - errors
 - packet 29
 - procedure cache 212
 - exceed logical page size 163
 - execute cursors
 - memory use of 330
 - execution 46
 - attributes 67
 - mixed workload precedence 86
 - precedence and users 87
 - ranking applications for 67
 - stored procedure precedence 87
 - system procedures for 71
 - execution class 67
 - attributes 69
 - predefined 68
 - user-defined 68
 - execution objects 67
 - behavior 64
 - performance hierarchy 67
 - scope 77
 - execution precedence
 - among applications 72
 - assigning 67
 - scheduling and 75
 - exp_row_size** option 189–194
 - create table** 190
 - default value 190
 - server-wide default 191
 - setting before **alter table...lock** 360
 - sp_chgattribute** 191
 - storage required by 269
 - expected row size. *See* **exp_row_size** option
 - expressions, maximum length 13
 - extents
 - allocation and **reservpagegap** 195
 - partitioned tables and extent stealing 114
 - space allocation and 156
- ## F
- fetch-and-discard cache strategy 175
 - fetching cursors
 - memory and 330
 - fillfactor
 - advantages of 184
 - disadvantages of 184
 - index creation and 183, 310

- index page size and 268
- locking and 202
- max_rows_per_page** compared to 203
- page splits and 184
- fillfactor** option
 - See also* **fillfactor** values
 - create index** 183
 - sorted_data** option and 188
- fillfactor** values
 - See also* **fillfactor** option
 - alter table...lock** 186
 - applied to data pages 187
 - applied to index pages 187
 - clustered index creation and 186
 - nonclustered index rebuilds 186
 - reorg rebuild** 186
 - table-level 186
- first normal form 137
 - See also* normalization
- first page
 - allocation page 158
 - text pointer 156
- fixed-length columns
 - calculating space for 254
 - data row size of 258, 264
 - for index keys 311
 - index row size and 259
 - overhead 311
- floating-point data xxii
- for load** option
 - performance and 348
- for update** option, **declare cursor**
 - optimizing and 341
- foreign keys
 - denormalization and 143
- formulas
 - cache hit ratio 219
 - table or index sizes 254–271
- forwarded rows
 - query on *systabstats* 193
 - reserve page gap and 194
- fragmentation, data
 - effects on asynchronous prefetch 371
 - page chain 371
- fragmentation, reserve page gap and 195
- free writes 50

G

- global allocation map (GAM) pages 157
- groups, number of for 12.5 13, 135

H

- Halloween problem
 - cursors and 332
- hardware
 - network 30
 - ports 34
 - terminology 92
- hash-based scans
 - asynchronous prefetch and 376
 - joins and 93
- header information
 - data pages 154
 - packet 21
 - “proc headers” 213
- heap tables 167–182
 - bcp** (bulk copy utility) and 354
 - delete operations 170
 - deletes and pages in cache 178
 - guidelines for using 180
 - I/O and 172
 - I/O inefficiency and 180
 - insert operations on 168
 - inserts and pages in cache 177
 - locking 169
 - maintaining 180
 - performance limits 169
 - select operations on 168, 176
 - updates and pages in cache 178
 - updates on 171
- high priority users 87
- historical data 147
- horizontal table splitting 147
- hot spots 87
 - binding caches to 220
- housekeeper free write percent** configuration
 - parameter 51
- housekeeper task 50–52
 - recovery time and 243

- I**
- I/O
- access problems and 91
 - asynchronous prefetch 363, ??–380
 - balancing load with segments 98
 - bcp** (bulk copy utility) and 355
 - buffer pools and 220
 - CPU and 53
 - create database** and 349
 - default caches and 174
 - devices and 90
 - disk 46
 - efficiency on heap tables 180
 - expected row size and 194
 - heap tables and 172
 - increasing size of 173
 - memory and 205
 - named caches and 220
 - network 45
 - parallel for **create database** 90
 - performance and 92
 - recovery interval and 351
 - select operations on heap tables and 176
 - server-wide and database 94
 - sp_spaceused** and 251
 - spreading between caches 388
 - transaction log and 182
- IDENTITY columns
- cursors and 332
 - indexing and performance 306
- image* datatype
- page size for storage 156
 - storage on separate device 98, 156
- index covering
- definition 152
- index keys, logical keys and 306
- index pages
- fillfactor effect on 185, 268
 - limiting number of rows on 203
 - page splits for 281
 - storage on 274
- index selection 308
- indexes 273–296
- access through 152, 273
 - bulk copy and 352
 - cache replacement policy for 230
 - choosing 152
 - computing number of pages 259
 - creating 344
 - cursors using 331
 - denormalization and 143
 - design considerations 297
 - dropping infrequently used 318
 - fillfactor and 183
 - guidelines for 310
 - intermediate level 276
 - leaf level 275
 - leaf pages 285
 - max_rows_per_page** and 203
 - number allowed 304
 - performance and 273–296
 - rebuilding 347
 - recovery and creation 345
 - root level 275
 - selectivity 299
 - size of 248
 - size of entries and performance 300
 - SMP environment and multiple 57
 - sort order changes 347
 - sp_spaceused** size report 251
 - temporary tables and 383, 391
 - types of 274
 - usefulness of 167
- indexing
- configure large buffer pools 320
 - create a clustered index first 320
- information (sp_sysmon)
- CPU usage 53
- initializing
- text or image pages 270
- insert operations
- clustered indexes 278
 - heap tables and 168
 - logging and 389
 - nonclustered indexes 289
 - page split exceptions and 280
 - partitions and 99
 - performance of 90
 - rebuilding indexes after many 347
- integer data
- in SQL xxii
- intermediate levels of indexes 276

isolation levels
 cursors 338

J

joins

choosing indexes for 307
 data cache and 176
 datatype compatibility in 311
 denormalization and 141
 derived columns instead of 144
 hash-based scan and 93
 normalization and 137
 temporary tables for 385

K

key values

index storage 273
 order for clustered indexes 277
 overflow pages and 282

keys, index

choosing columns for 306
 clustered and nonclustered indexes and 274
 composite 312
 logical keys and 306
 monotonically increasing 281
 size and performance 310
 size of 304
 unique 310

L

large I/O

asynchronous prefetch and 374
 named data caches and 225

large object (LOB) 98

leaf levels of indexes 275

fillfactor and number of rows 268
 queries on 153
 row size calculation 261, 265

leaf pages 285

calculating number in index 262, 266

limiting number of rows on 203

levels

indexes 275
 tuning 5–10

lightweight process 37

listeners, network 34

load balancing for partitioned tables 114

maintaining 131

local backups 350

locking 16–??

create index and 344

heap tables and inserts 169

last page inserts and 306

tempdb and 388

worktables and 388

log I/O size

matching 227

tuning 224

using large 235

logging

bulk copy and 352

minimizing in *tempdb* 389

logical database design 133, 150

logical device name 92

logical expressions xxii

logical keys, index keys and 306

logical page sizes 12, 134, 153, 154

logical process manager 67

logins 45

number of for 12.5 13, 135

look-ahead set 364

dbcc and 367

during recovery 366

nonclustered indexes and 367

sequential scans and 366

lookup tables, cache replacement policy for 230

LRU replacement strategy 174

M

maintenance tasks 343–355

performance and 90

managing denormalized data 148

map, object allocation. *See* object allocation map (OAM)

pages

Index

- matching index scans 292
 - max async i/os per engine** configuration parameter
 - asynchronous prefetch and 370
 - max async i/os per server** configuration parameter
 - asynchronous prefetch and 370
 - max_rows_per_page** option
 - fillfactor** compared to 203
 - locking and 202
 - select into** effects 204
 - memory
 - cursors and 328
 - how to allocate 210
 - I/O and 205
 - named data caches and 220
 - network packets and 28
 - performance and 205–245
 - shared 44
 - messages
 - See also* errors
 - mixed workload execution priorities 86
 - model, SMP process 43
 - modes of disk mirroring 96
 - monitoring
 - CPU usage 53
 - data cache performance 218
 - index usage 318
 - network activity 29
 - performance 5
 - monitoring environment 66
 - Monitoring indexes
 - examples of 309
 - using **sp_monitorconfig** 308
 - monitoring indexes ??–310
 - MRU replacement strategy 174
 - asynchronous prefetch and 376
 - multicolumn index. *See* composite indexes
 - multiple network engines 45
 - multiple network listeners 34
 - multitasking 39
 - multithreading 35
- ## N
- nesting
 - temporary tables and 391
 - network engines 45
 - network I/O 45
 - network packets
 - global variables 29
 - sp_monitor** system procedure 29, 53
 - networks 19
 - cursor activity of 336
 - hardware for 30
 - multiple listeners 34
 - performance and 19–34
 - ports 34
 - reducing traffic on 30, 355
 - server based techniques 30
 - nonclustered indexes 274
 - asynchronous prefetch and 367
 - definition of 285
 - delete operations 290
 - estimating size of 261–263
 - guidelines for 307
 - insert operations 289
 - number allowed 304
 - select operations 287
 - size of 251, 261, 265, 285
 - structure 286
 - nonleaf rows 262
 - nonmatching index scans 293–294
 - normal forms 15
 - normalization 135
 - first normal form 137
 - joins and 137
 - second normal form 138
 - temporary tables and 385
 - third normal form 139
 - null columns
 - storage of rows 155
 - storage size 256
 - variable-length 310
 - null values
 - datatypes allowing 310
 - text* and *image* columns 270
 - number (quantity of)
 - bytes per index key 304
 - clustered indexes 274
 - cursor rows 341
 - indexes per table 304
 - nonclustered indexes 274

- OAM pages 263, 267
- packet errors 29
- procedure (“proc”) buffers 213
- processes 38
- rows (rowtotal), estimated 250
- rows on a page 203
- number of columns and sizes 161
- number of groups 13, 135
- number of logins 13, 135
- number of sort buffers 320
- number of users 13, 135
- numbers
 - row offset 285
- numeric expressions xxii

O

- object allocation map (OAM) pages 158
 - overhead calculation and 260, 265
- object allocation mapp (OAM) pages
 - LRU strategy in data cache 175
- object size
 - viewing with **optdiag** 249
- offset table
 - nonclustered index selects and 287
 - row IDs and 285
 - size of 155
- online backups 351
- online transaction processing (OLTP)
 - execution preference assignments 86
 - named data caches for 221
 - network packet size for 27
- open** command
 - memory and 330
- optdiag** utility command
 - object sizes and 249
- optimization
 - cursors 330
- optimizer
 - cache strategies and 228
 - dropping indexes not used by 318
 - indexes and 297
 - nonunique entries and 299
 - temporary tables and 390
- OR strategy

- cursors and 340
- order
 - composite indexes and 312
 - data and index storage 274
 - index key values 277
 - presorted data and index creation 345
 - recovery of databases 351
 - result sets and performance 180
- order by** clause
 - indexes and 273
- output
 - sp_estspace** 300
 - sp_spaceused** 250
- overflow pages 282
 - key values and 282
- overhead
 - calculation (space allocation) 263, 267
 - clustered indexes and 180
 - cursors 336
 - datatypes and 310, 319
 - network packets and 28
 - nonclustered indexes 311
 - object size calculations 254
 - pool configuration 239
 - row and page 254
 - single process 37
 - space allocation calculation 260, 265
 - variable-length and null columns 256
 - variable-length columns 311
- overheads 160

P

- @@pack_received** global variable 29
- @@pack_sent** global variable 29
- packet size 27
- @@packet_errors** global variable 29
- packets
 - default 28
 - number 28
 - size specification 29
- packets, network 21
 - size, configuring 27
- page chain kinks
 - asynchronous prefetch and 371, 378

Index

- clustered indexes and 379
- defined 371
- heap tables and 379
- nonclustered indexes and 379
- page chains
 - overflow pages and 282
 - placement 90
 - text or image* data 270
 - unpartitioning 107
- page splits
 - data pages 279
 - fillfactor effect on 184
 - index pages and 281
 - max_rows_per_page** setting and 203
 - nonclustered indexes, effect on 279
 - object size and 249
 - performance impact of 281
 - reducing 184
- page utilization percent** configuration parameter
 - object size estimation and 255
- pages
 - global allocation map (GAM) 157
 - overflow 282
- pages, control
 - updating statistics on 115
- pages, data 153–181
 - bulk copy and allocations 352
 - calculating number of 258, 264
 - fillfactor effect on 268
 - fillfactor for SMP systems 58
 - linking 167
 - size 153
 - splitting 279
- pages, index
 - aging in data cache 216
 - calculating number of 259
 - calculating number of non-leaf 266
 - fillfactor effect on 185, 268
 - fillfactor for SMP systems 58
 - leaf level 285
 - storage on 274
- pages, OAM (Object Allocation Map)
 - number of 263
- pages, OAM (object allocation map) 158
 - aging in data cache 216
 - number of 260, 265, 267
- parallel query processing
 - asynchronous prefetch and 376
 - object placement and 90
 - performance of 91
- parallel sort
 - configure enough sort buffers 320
- partition** clause, **alter table** command 106
- partition-based scans
 - asynchronous prefetch and 377
- partitioned tables 99
 - bcp** (bulk copy utility) and 110, 354
 - changing the number of partitions 107
 - command summary 106
 - configuration parameters for 102
 - configuration parameters for indexing 108
 - create index** and 108
 - creating new 117
 - data distribution in 111
 - devices and 114, 123, 128
 - distributing data across 108, 120
 - extent stealing and 114
 - information on 111
 - load balancing and 114
 - loading with **bcp** 110
 - maintaining 116, 131
 - moving with **on segmentname** 119
 - read-mostly 105
 - read-only 104
 - segment distribution of 102
 - size of 111, 115
 - sorted data** option and 118
 - space planning for 103
 - statistics 116
 - statistics updates 115
 - unpartitioning 107
 - updates and 105
 - updating statistics 116
- partitioning tables 106
- partitions
 - ratio of sizes 111
 - size of 111, 115
- performance 3
 - analysis 14
 - backups and 351
 - bcp** (bulk copy utility) and 353
 - cache hit ratio 218

- clustered indexes and 180
 - designing 4
 - indexes and 297
 - networks 19
 - number of indexes and 299
 - problems 19
 - techniques 20
 - tempdb* and 381–391
 - performing benchmark tests 65
 - performing disk I/O 46
 - physical device name 92
 - point query 152
 - pointers
 - index 274
 - last page, for heap tables 168
 - page chain 167
 - text and image page 156
 - pools, data cache
 - configuring for operations on heap tables 173
 - large I/Os and 225
 - overhead 239
 - ports, multiple 34
 - precedence
 - rule (execution hierarchy) 77
 - precedence rule, execution hierarchy 78
 - precision, datatype
 - size and 256
 - predefined execution class 68
 - prefetch
 - asynchronous 363–??
 - sequential 173
 - primary key** constraint
 - index created by 304
 - primary keys
 - normalization and 138
 - splitting tables and 146
 - priority 69
 - application 67
 - assigning 68
 - precedence rule 78
 - run queues 75
 - task 67
 - “proc headers” 213
 - procedure (“proc”) buffers 213
 - procedure cache
 - cache hit ratio 213
 - errors 213
 - query plans in 212
 - size report 212
 - sizing 213
 - procedure cache sizing** configuration parameter 211
 - process model 43
 - processes (server tasks) 39
 - identifier (PID) 38
 - lightweight 37
 - number of 38
 - overhead 37
 - run queue 39
 - ptn_data_pgs** system function 115
- ## Q
- queries
 - point 152
 - range 299
 - unindexed columns in 153
 - query plans
 - procedure cache storage 212
 - unused and procedure cache 212
 - updatable cursors and 340
 - query processing
 - large I/O for 227
 - queues
 - run 46
 - scheduling and 40
 - sleep 40
- ## R
- RAID devices
 - partitioned tables and 103
 - range queries 299
 - read-only cursors 331
 - indexes and 331
 - locking and 336
 - reads
 - clustered indexes and 278
 - disk mirroring and 96
 - image* values 156
 - named data caches and 241

Index

- text* values 156
- recompilation
 - cache binding and 240
- recovery
 - asynchronous prefetch and 366
 - configuring asynchronous prefetch for 377
 - housekeeper task and 51
 - index creation and 345
 - log placement and speed 95
- recovery interval in minutes** configuration parameter 216, 242
 - I/O and 351
- re-creating
 - indexes 108, 345
- referential integrity
 - references** and unique index requirements 310
- relaxed LRU replacement policy
 - indexes 230
 - lookup tables 230
 - transaction logs 230
- remote backups 350
- replacement policy. *See* cache replacement policy
- replacement strategy. *See* LRU replacement strategy; MRU replacement strategy
- replication
 - network activity from 31
 - tuning levels and 6
- reports
 - procedure cache size 212
 - sp_estspace** 252
- reserved pages, **sp_spaceused** report on 252
- reservepagegap** option 195–200
 - cluster ratios 194, 199
 - create index** 197
 - create table** 196
 - extent allocation and 195
 - forwarded rows and 194
 - sp_chgattribute** 197
 - space usage and 194
 - storage required by 269
- response time
 - definition of 3
 - other users affecting 32
 - table scans and 152
- risks of denormalization 142
- root level of indexes 275

- rounding
 - object size calculation and 254
- row ID (RID) 285
- row offset number 285
- rows per data page 165
- rows, index
 - size of leaf 261, 265
 - size of non-leaf 262
- rows, table
 - splitting 148
- run queue 38, 39, 46

S

- scans, table
 - avoiding 273
 - performance issues 152
- scheduling, Server engines 44
 - tasks 40
- scope rule 77, 79
- search conditions
 - clustered indexes and 306
- second normal form 138
 - See also* normalization
- segments 92
 - changing table locking schemes 358
 - clustered indexes on 98
 - database object placement on 93, 98
 - free pages in 114
 - moving tables between 119
 - nonclustered indexes on 98
 - partition distribution over 102
 - tempdb* 386
- select *** command
 - logging of 389
- select** command
 - optimizing 299
- select into** command
 - heap tables and 169
 - large I/O for 227
- select operations
 - clustered indexes and 277
 - heaps 168
 - nonclustered indexes 287

- sequential prefetch 173, 225
- server
 - other tools 30
- servers
 - scheduler 42
 - uniprocessor and SMP 57
- set theory operations
 - compared to row-oriented programming 326
- shared** keyword
 - cursors and 331
- shared locks
 - read-only cursors 331
- single CPU 38
- single-process overhead 37
- size
 - data pages 153
 - datatypes with precisions 256
 - formulas for tables or indexes 254–271
 - I/O 173, 225
 - indexes 248
 - nonclustered and clustered indexes 285
 - object (**sp_spaceused**) 250
 - partitions 111
 - predicting tables and indexes 257–271
 - procedure cache 212, 213
 - sp_spaceused** estimation 252
 - stored procedure 214
 - tables 248
 - tempdb* database 384
 - triggers 214
 - views 214
- skew in partitioned tables
 - information on 111
- sleep queue 40
- SMP (symmetric multiprocessing) systems
 - application design in 57
 - architecture 43
 - disk management in 58
 - named data caches for 222
 - temporary tables and 58
- sort operations (**order by**)
 - improving performance of 344
 - indexing to avoid 273
 - performance problems 382
- sort order
 - rebuilding indexes after changing 347
- sorted data, reindexing 345, 348
- sorted_data** option
 - fillfactor** and 188
 - reservepagegap** and 200
- sorted_data** option, **create index**
 - partitioned tables and 118
 - sort suppression and 345
- sp_addengine** system procedure 73
- sp_addextclass** system procedure 68
- sp_bindextclass** system procedure 68
- sp_chgattribute** system procedure
 - exp_row_size** 191
 - fillfactor** 185–188
 - reservepagegap** 197
- sp_estspace** system procedure
 - advantages of 253
 - disadvantages of 254
 - planning future growth with 252
- sp_help** system procedure
 - displaying expected row size 192
- sp_helppartition** system procedure 111
- sp_helpsegment** system procedure
 - checking data distribution 114
- sp_logiosize** system procedure 235
- sp_monitor** system procedure 53
 - network packets 29
- sp_spaceused** system procedure 250
 - row total estimate reported 250
- space 160, 161
 - clustered compared to nonclustered indexes 285
 - estimating table and index size 257–271
 - extents 156
 - for *text* or *image* storage 156
 - reclaiming 180
 - unused 156
- space allocation
 - clustered index creation 304
 - contiguous 159
 - deallocation of index pages 285
 - deletes and 171
 - extents 156
 - index page splits 281
 - monotonically increasing key values and 281
 - object allocation map (OAM) pages 260, 265
 - overhead calculation 260, 263, 265, 267
 - page splits and 279

Index

- predicting tables and indexes 257–271
 - procedure cache 213
 - sp_spaceused** 252
 - tempdb* 387
 - unused space within 156
 - space management properties 183–204
 - object size and 268
 - reserve page gap 195–200
 - space usage 360
 - speed (server)
 - memory compared to disk 205
 - select into** 389
 - sort operations 344
 - spinlocks
 - contention 232
 - data caches and 221
 - splitting
 - data pages on inserts 279
 - horizontal 147
 - tables 146
 - vertical 148
 - SQL standards
 - cursors and 326
 - steps
 - problem analysis 14
 - storage management
 - collapsed tables effect on 145
 - delete operations and 170
 - I/O contention avoidance 93
 - page proximity 159
 - row storage 155
 - space deallocation and 284
 - store procedures, maximum length 13
 - stored procedures
 - cursors within 334
 - hot spots and 87
 - performance and 90
 - procedure cache and 212
 - size estimation 214
 - temporary tables and 392
 - striping *tempdb* 384
 - subprocesses 39
 - switching context 39
 - sybsecurity* database
 - audit queue and 244
 - placement 94
 - symbols
 - in SQL statements xx
 - Symmetric Multi Processing System. *See* SMP 44
 - sysgams* table 157
 - sysindexes* table
 - data access and 160
 - text objects listed in 156
 - sysprocedures* table
 - query plans in 212
 - system tables
 - data access and 160
 - performance and 90
- ## T
- table scans
 - asynchronous prefetch and 366
 - avoiding 273
 - performance issues 152
 - tables
 - collapsing 145
 - denormalizing by splitting 146
 - designing 135
 - duplicating 146
 - estimating size of 254
 - heap 167–182
 - moving with **on segmentname** 119
 - normal in *tempdb* 383
 - normalization 135
 - partitioning 99, 106
 - secondary 319
 - size of 248
 - size with a clustered index 257, 263
 - unpartitioning 107
 - tabular data stream 21
 - tabular data stream (TDS) protocol 21
 - task level tuning
 - algorithm 59
 - tasks
 - client 36
 - execution 46
 - queued 40
 - scheduling 40
 - TDS. *See* Tabular Data Stream
 - tempdb* database

- data caches 387
- logging in 389
- named caches and 221
- performance and 381–391
- placement 94, 386
- segments 386
- in SMP environment 58
- space allocation 387
- striping 384
- temporary tables
 - denormalization and 385
 - indexing 391
 - nesting procedures and 391
 - normalization and 385
 - optimizing 390
 - performance considerations 90, 382
 - permanent 383
 - SMP systems 58
- testing
 - data cache performance 218
 - “hot spots” 307
 - nonclustered indexes 311
- text* datatype
 - chain of text pages 270
 - page size for storage 156
 - storage on separate device 98, 156
 - sysindexes* table and 156
- third normal form. *See* normalization
- thresholds
 - bulk copy and 353
 - database dumps and 351
- throughput 4
 - measuring for devices 103
- time interval
 - recovery 242
 - since **sp_monitor** last run 53
- time slice** 69
 - configuration parameter 42
- time slice** configuration parameter
 - CPU yields and 43
- tools
 - packet monitoring with **sp_monitor** 29
- transaction length 58
- transaction logs
 - cache replacement policy for 230
 - log I/O size and 234
- named cache binding 221
- placing on separate segment 94
- on same device 95
- storage as heap 181
- transactions
 - logging and 389
- triggers
 - managing denormalized data with 149
 - procedure cache and 212
 - size estimation 214
- truncate table** command
 - not allowed on partitioned tables 102
- tuning
 - Adaptive Server layer 7
 - application layer 6
 - asynchronous prefetch 373
 - database layer 6
 - definition of 4
 - devices layer 8
 - hardware layer 9
 - levels 5–10
 - network layer 8
 - operating system layer 9
 - recovery interval 242
- two-phase commit
 - network activity from 31

U

- union** operator
 - cursors and 340
- uniprocessor system 38
- unique constraints
 - index created by 304
- unique indexes 273
 - optimizing 310
- units, allocation. *See* allocation units
- unpartition** clause, **alter table** 107
- unpartitioning tables 107
- unused space
 - allocations and 156
- update** command
 - image* data and 270
 - text* data and 270
- update cursors 331

Index

- update locks
 - cursors and 331
- update operations
 - heap tables and 171
 - index updates and 311
- update partition statistics** command 115, 116
- update statistics** command
 - large I/O for 227
- user connections
 - network packets and 28
- user log cache (ULC)
 - log size and 234
- user log cache, in ALS 48, 322
- user-defined execution class 68
- users
 - assigning execution priority 87
 - login information 45
- users, number of for 12.5 13, 135
- Using Asynchronous log service 47, 320
- Using Asynchronous log service, ALS 47, 320

- tempdb* and 384
- write operations
 - disk mirroring and 96
 - free 50
 - housekeeper process and 52
 - image* values 156
 - serial mode of disk mirroring 96
 - text* values 156

Y

- yields, CPU
 - cpu grace time** configuration parameter 43
 - time slice** configuration parameter 43
 - yield points 42

V

- variable-length 163
- variable-length columns
 - index overhead and 319
- variables, maximum length 13
- vertical table splitting 148
- views
 - collapsing tables and 146
 - size estimation 214

W

- wash area 215
 - configuring 238
- wash marker 174
- when to use ALS 48, 322
- where** clause
 - creating indexes for 307
 - table scans and 167
- worker processes 36
- worktables
 - locking and 388