



Sybase® Adaptive Server™ Enterprise
Performance and Tuning Guide

Adaptive Server™

Document ID: 32645-01-1150-02

September 1997

Copyright Information

Copyright © 1989–1997 by Sybase, Inc. All rights reserved.

Sybase, Inc., 6475 Christie Avenue, Emeryville, CA 94608.

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. 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.

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, the Sybase logo, APT-FORMS, Certified SYBASE Professional, Data Workbench, First Impression, InfoMaker, PowerBuilder, Powersoft, Replication Server, S-Designer, SQL Advantage, SQL Debug, SQL SMART, SQL Solutions, Transact-SQL, VisualWriter, and VQL are registered trademarks of Sybase, Inc. Adaptable Windowing Environment, Adaptive Component Architecture, Adaptive Server, Adaptive Server Monitor, ADA Workbench, AnswerBase, Application Manager, AppModeler, APT-Build, APT-Edit, APT-Execute, APT-Library, APT-Translator, APT Workbench, Backup Server, BayCam, Bit-Wise, ClearConnect, Client-Library, Client Services, CodeBank, Column Design, Connection Manager, DataArchitect, Database Analyzer, DataExpress, Data Pipeline, DataWindow, DB-Library, dbQ, Developers Workbench, DirectConnect, Distribution Agent, Distribution Director, Dynamo, Embedded SQL, EMS, Enterprise Client/Server, Enterprise Connect, Enterprise Manager, Enterprise SQL Server Manager, Enterprise Work Architecture, Enterprise Work Designer, Enterprise Work Modeler, EWA, Formula One, Gateway Manager, GeoPoint, ImpactNow, InformationConnect, InstaHelp, InternetBuilder, iScript, Jaguar CTS, jConnect for JDBC, KnowledgeBase, Logical Memory Manager, MainframeConnect, Maintenance Express, MAP, MDI Access Server, MDI Database Gateway, media.splash, MetaWorks, MethodSet, Net-Gateway, NetImpact, Net-Library, ObjectConnect, ObjectCycle, OmniConnect, OmniSQL Access Module, OmniSQL Toolkit, 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 DB-Net, PC Net Library, Power++, Power AMC, PowerBuilt, PowerBuilt with PowerBuilder, PowerDesigner, Power J, PowerScript, PowerSite, PowerSocket, Powersoft Portfolio, Power Through Knowledge, PowerWare Desktop, PowerWare Enterprise, ProcessAnalyst, Quickstart Datamart, Replication Agent, Replication Driver, Replication Server Manager, Report-Execute, Report Workbench, Resource Manager, RW-DisplayLib, RW-Library, SAFE, SDF, Secure SQL Server, Secure SQL Toolset, Security Guardian, SKILLS, smart.partners, smart.parts, smart.script, SQL Anywhere, SQL Central, SQL Code Checker, SQL Edit, SQL Edit/TPU, SQL Modeler, SQL Remote, SQL Server, SQL Server/CFT, SQL Server/DBM, SQL Server Manager, SQL Server SNMP SubAgent, SQL Station, SQL Toolset, Sybase Client/Server Interfaces, Sybase Development Framework, Sybase Gateways, Sybase IQ, Sybase MPP, Sybase SQL Desktop, Sybase SQL Lifecycle, Sybase SQL Workgroup, Sybase Synergy Program, Sybase Virtual Server Architecture, Sybase User Workbench, SybaseWare, SyBooks, System 10, System 11, the System XI logo, SystemTools, Tabular Data Stream, The Architecture for Change, The Enterprise Client/Server Company, The Model for Client/Server Solutions, The Online Information Center, Translation Toolkit, Turning Imagination Into Reality, Unibom, Unilib, Uninull, Unisep, Unistring, Viewer, Visual Components, VisualSpeller, WarehouseArchitect, WarehouseNow, Warehouse WORKS, Watcom, Watcom SQL, Watcom SQL Server, Web.SQL, WebSights, WebViewer, WorkGroup SQL Server, XA-Library, and XA-Server are trademarks of Sybase, Inc. 6/97

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

Table of Contents

About This Book

Audience	xlvi
How to Use This Book	xlvi
Adaptive Server Enterprise Documents	xlvii
Other Sources of Information	xlviii
Conventions	xlix
Formatting SQL Statements	xlix
Font and Syntax Conventions	xlix
Case	l
Expressions	li
Examples	li
If You Need Help	lii

1. Introduction to Performance Analysis

What Is “Good Performance”?	1-1
Response Time	1-1
Throughput	1-1
Designing for Performance	1-2
What Is Tuning?	1-2
Tuning Levels	1-3
Application Layer	1-3
Database Layer	1-4
Adaptive Server Layer	1-5
Devices Layer	1-5
Network Layer	1-6
Hardware Layer	1-6
Operating System Layer	1-7
Know the System Limits	1-7
Know Your Tuning Goals	1-7
Steps in Performance Analysis	1-7
Using <i>sp_sysmon</i> to Monitor Performance	1-9

2. Database Design and Denormalizing for Performance

Database Design	2-1
Physical Database Design for Adaptive Server	2-2

Normalization	2-3
Levels of Normalization	2-3
Benefits of Normalization	2-4
First Normal Form	2-4
Second Normal Form	2-5
Third Normal Form	2-6
Denormalizing for Performance	2-8
Risks of Denormalization	2-8
Disadvantages of Denormalization	2-9
Performance Advantages of Denormalization	2-9
Denormalization Input	2-10
Denormalization Techniques	2-10
Adding Redundant Columns	2-11
Adding Derived Columns	2-11
Collapsing Tables	2-12
Duplicating Tables	2-13
Splitting Tables	2-14
Horizontal Splitting	2-14
Vertical Splitting	2-15
Managing Denormalized Data	2-16
Using Triggers to Manage Denormalized Data	2-17
Using Application Logic to Manage Denormalized Data	2-17
Batch Reconciliation	2-18
3. Data Storage	
Major Performance Gains Through Query Optimization	3-1
Query Processing and Page Reads	3-2
Adaptive Server Data Pages	3-3
Row Density on Data Pages	3-4
Extents	3-5
Linked Data Pages	3-5
Text and Image Pages	3-6
Additional Page Types	3-7
Global Allocation Map (GAM) Pages	3-8
Allocation Pages	3-8
Object Allocation Map (OAM) Pages	3-9
Why the Range?	3-9
Relationships Between Objects, OAM Pages, and Allocation Pages	3-9
The <i>sysindexes</i> Table and Data Access	3-11
Heaps of Data: Tables Without Clustered Indexes	3-11

Select Operations on Heaps	3-12
Inserting Data into a Heap	3-12
Deleting Data from a Heap	3-14
Update Operations on Heaps	3-14
How Adaptive Server Performs I/O for Heap Operations	3-15
Sequential Prefetch, or Large I/O	3-15
Caches and Object Bindings	3-16
Heaps, I/O, and Cache Strategies	3-16
Overview of Cache Strategies	3-16
LRU Replacement Strategy	3-16
When LRU Strategy Is Used	3-17
MRU Replacement Strategy	3-18
Select Operations and Caching	3-19
Data Modification and Caching	3-19
Caching and Inserts on Heaps	3-19
Caching and Update and Delete Operations on Heaps	3-20
Heaps: Pros and Cons	3-21
Guidelines for Using Heaps	3-21
Maintaining Heaps	3-21
Methods for Maintaining Heaps	3-21
Reclaiming Space by Creating a Clustered Index	3-22
Reclaiming Space Using <i>bcp</i>	3-22
The Transaction Log: A Special Heap Table	3-22

4. How Indexes Work

From Heaps of Pages to Fast Performance	4-1
What Are Indexes?	4-2
Types of Indexes	4-2
Index Pages	4-2
Root Level	4-3
Leaf Level	4-3
Intermediate Level	4-3
Clustered Indexes	4-4
Clustered Indexes and Select Operations	4-5
Clustered Indexes and Insert Operations	4-7
Page Splitting on Full Data Pages	4-7
Exceptions to Page Splitting	4-9
Page Splitting on Index Pages	4-9
Performance Impacts of Page Splitting	4-9
Overflow Pages	4-10
Clustered Indexes and Delete Operations	4-10

Deleting the Last Row on a Page	4-11
Index Page Merges	4-13
Nonclustered Indexes	4-13
Leaf Pages Revisited	4-14
Row IDs and the Offset Table	4-14
Nonclustered Index Structure	4-16
Nonclustered Indexes and Select Operations	4-17
Nonclustered Index Performance	4-18
Nonclustered Indexes and Insert Operations	4-19
Nonclustered Indexes and Delete Operations	4-20
Index Covering	4-21
Matching Index Scans	4-21
Nonmatching Index Scans	4-22
Indexes and Caching	4-23
Using Separate Caches for Data and Index Pages	4-25
Index Trips Through the Cache	4-26
5. Locking in Adaptive Server	
How Locking Affects Performance	5-1
Overview of Locking	5-2
Granularity of Locks	5-3
Types of Locks in Adaptive Server	5-3
Page Locks	5-4
Table Locks	5-6
Demand Locks	5-7
Demand Locking with Serial Execution	5-8
Demand Locking with Parallel Execution	5-9
Summary of Lock Types	5-11
Lock Compatibility	5-12
How Isolation Levels Affect Locking	5-12
Isolation Level 0	5-12
Isolation Level 1	5-14
Isolation Level 2	5-15
Isolation Level 3	5-16
Adaptive Server Default Isolation Level	5-18
Controlling Isolation Levels	5-19
Syntax for Query Level Locking Options	5-19
Setting Isolation Levels for a Session	5-20
Using <i>holdlock</i> , <i>noholdlock</i> , or <i>shared</i>	5-20
Using the <i>at isolation</i> Clause	5-21
Making Locks More Restrictive	5-21

Using <i>read committed</i>	5-22
Making Locks Less Restrictive	5-23
Using <i>read uncommitted</i>	5-23
Using <i>shared</i>	5-23
Examples of Locking and Isolation Levels	5-23
Cursors and Locking	5-26
Using the <i>shared</i> Keyword	5-27
Deadlocks and Concurrency	5-28
Avoiding Deadlocks	5-30
Acquire Locks on Objects in the Same Order	5-30
Delaying Deadlock Checking	5-31
Locking and Performance	5-32
Using <i>sp_sysmon</i> While Reducing Lock Contention	5-32
Reducing Lock Contention	5-33
Keeping Transactions Short	5-33
Avoiding “Hot Spots”	5-34
Decreasing the Number of Rows per Page	5-35
Additional Locking Guidelines	5-37
Reporting on Locks and Locking Behavior	5-38
Getting Information About Blocked Processes with <i>sp_who</i>	5-38
Viewing Locks with <i>sp_lock</i>	5-38
Viewing Locks with <i>sp_familylock</i>	5-40
Printing Deadlock Information to the Error Log	5-40
Observing Deadlocks Among Serial Queries	5-41
Observing Deadlocks Among Worker Processes	5-42
Observing Locks with <i>sp_sysmon</i>	5-45
Intrafamily Blocking During Network Buffer Merges	5-45
Configuring Locks and Lock Promotion Thresholds	5-46
Configuring Adaptive Server’s Lock Limit	5-46
Setting the Lock Promotion Thresholds	5-47
Lock Promotion Occurs on a Per-Session Basis	5-47
<i>lock promotion HWM</i>	5-48
<i>lock promotion LWM</i>	5-48
<i>lock promotion PCT</i>	5-49
Setting Lock Promotion Thresholds Server-Wide	5-50
Setting the Lock Promotion Threshold for a Table or Database ...	5-50
Precedence of Settings	5-50
Dropping Database and Table Settings	5-51
Using <i>sp_sysmon</i> While Tuning Lock Promotion Thresholds	5-51

6. Determining or Estimating the Sizes of Tables and Indexes

Tools for Determining the Sizes of Tables and Indexes	6-1
Why Should You Care About the Sizes of Objects?	6-1
System Table Sizes	6-2
Effects of Data Modifications on Object Sizes	6-3
OAM Pages and Size Statistics	6-3
Using <i>sp_spaceused</i> to Display Object Size	6-4
Advantages of <i>sp_spaceused</i>	6-6
Disadvantages of <i>sp_spaceused</i>	6-6
Using <i>dbcc</i> to Display Object Sizes	6-6
Advantages of <i>dbcc</i>	6-9
Disadvantages of <i>dbcc</i>	6-9
Using <i>sp_estspace</i> to Estimate Object Size	6-9
Advantages of <i>sp_estspace</i>	6-11
Disadvantages of <i>sp_estspace</i>	6-11
Using Formulas to Estimate Object Size	6-11
Factors That Can Affect Storage Size	6-12
Storage Sizes for Datatypes	6-13
Calculating the Sizes of Tables and Clustered Indexes	6-14
Step 1: Calculate the Data Row Size	6-14
Step 2: Compute the Number of Data Pages	6-15
Step 3: Compute the Size of Clustered Index Rows	6-15
Step 4: Compute the Number of Clustered Index Pages	6-16
Step 5: Compute the Total Number of Index Pages	6-17
Step 6: Calculate Allocation Overhead and Total Pages	6-17
Example: Calculating the Size of a 9,000,000-Row Table	6-18
Calculating the Data Row Size (Step 1)	6-18
Calculating the Number of Data Pages (Step 2)	6-19
Calculating the Clustered Index Row Size (Step 3)	6-19
Calculating the Number of Clustered Index Pages (Step 4)	6-19
Calculating the Total Number of Index Pages (Step 5)	6-19
Calculating the Number of OAM Pages and Total Pages (Step 6) ..	6-19
Calculating the Size of Nonclustered Indexes	6-20
Step 7: Calculate the Size of the Leaf Index Row	6-20
Step 8: Calculate the Number of Leaf Pages in the Index	6-21
Step 9: Calculate the Size of the Non-Leaf Rows	6-21
Step 10: Calculate the Number of Non-Leaf Pages	6-21
Step 11: Calculate the Total Number of Non-Leaf Index Pages	6-22
Step 12: Calculate Allocation Overhead and Total Pages	6-22
Example: Calculating the Size of a Nonclustered Index	6-23
Calculate the Size of the Leaf Index Row (Step 7)	6-23

Calculate the Number of Leaf Pages (Step 8)	6-23
Calculate the Size of the Non-Leaf Rows (Step 9)	6-24
Calculate the Number of Non-Leaf Pages (Step 10)	6-24
Calculating Totals (Step 11)	6-24
Calculating OAM Pages Needed (Step 12)	6-24
Total Pages Needed	6-25
Other Factors Affecting Object Size	6-25
Effects of Setting <i>fillfactor</i> to 100 Percent	6-25
Other <i>fillfactor</i> Values	6-26
Distribution Pages	6-26
Using Average Sizes for Variable Fields	6-26
Very Small Rows	6-28
<i>max_rows_per_page</i> Value	6-28
<i>text</i> and <i>image</i> Data Pages	6-28
Advantages of Using Formulas to Estimate Object Size	6-29
Disadvantages of Using Formulas to Estimate Object Size	6-29

7. Indexing for Performance

How Indexes Can Affect Performance	7-1
Symptoms of Poor Indexing	7-2
Detecting Indexing Problems	7-3
Lack of Indexes Is Causing Table Scans	7-3
Index Is Not Selective Enough	7-3
Index Does Not Support Range Queries	7-4
Too Many Indexes Slow Data Modification	7-4
Index Entries Are Too Large	7-4
Index Limits and Requirements	7-6
Tools for Query Analysis and Tuning	7-6
Using <i>sp_sysmon</i> to Observe the Effects of Index Tuning	7-9
Indexes and I/O Statistics	7-9
Total Actual I/O Cost Value	7-10
Statistics for Writes	7-10
Statistics for Reads	7-11
Sample Output With and Without an Index	7-11
Scan Count	7-12
Queries Reporting a Scan Count of 1	7-13
Queries Reporting a Scan Count of More Than 1	7-13
Queries Reporting Scan Count of 0	7-14
Relationship Between Physical and Logical Reads	7-15
Logical Reads, Physical Reads, and 2K I/O	7-16

Physical Reads and Large I/O	7-16
Reads and Writes on Worktables	7-16
Effects of Caching on Reads	7-16
Estimating I/O	7-17
Table Scans	7-17
Evaluating the Cost of a Table Scan	7-18
Evaluating the Cost of Index Access	7-19
Evaluating the Cost of a Point Query	7-20
Evaluating the Cost of a Range Query	7-20
Range Queries with Covering Nonclustered Indexes	7-21
Range Queries with Noncovering Nonclustered Indexes	7-23
Optimizing Indexes for Queries That Perform Sorts	7-25
How the Optimizer Costs Sort Operations	7-25
Sorts and Clustered Indexes	7-26
Sorts and Nonclustered Indexes	7-28
Sorts When the Index Covers the Query	7-29
Descending Scans and Sorts	7-29
Descending Scans and Joins	7-29
When Sorts Are Still Needed	7-30
Deadlocks and Descending Scans	7-30
Choosing Indexes	7-30
Index Keys and Logical Keys	7-31
Guidelines for Clustered Indexes	7-31
Choosing Clustered Indexes	7-32
Candidates for Nonclustered Indexes	7-33
Other Indexing Guidelines	7-33
Choosing Nonclustered Indexes	7-34
Performance Price for Data Modification	7-35
Choosing Composite Indexes	7-35
User Perceptions and Covered Queries	7-36
The Importance of Order in Composite Indexes	7-37
Advantages of Composite Indexes	7-37
Disadvantages of Composite Indexes	7-38
Key Size and Index Size	7-38
Techniques for Choosing Indexes	7-39
Choosing an Index for a Range Query	7-39
Adding a Point Query with Different Indexing Requirements	7-40
Index Statistics	7-42
The Distribution Table	7-42
The Density Table	7-44
How the Optimizer Uses the Statistics	7-45

How the Optimizer Uses the Distribution Table	7-46
How the Optimizer Uses the Density Table	7-46
Index Maintenance	7-47
Monitoring Applications and Indexes Over Time	7-47
Dropping Indexes That Hurt Performance	7-47
Maintaining Index Statistics	7-47
Rebuilding Indexes	7-48
Speeding Index Creation with <i>sorted_data</i>	7-49
Displaying Information About Indexes	7-49
Tips and Tricks for Indexes	7-50
Creating Artificial Columns	7-51
Keeping Index Entries Short and Avoiding Overhead	7-51
Dropping and Rebuilding Indexes	7-51
Choosing Fillfactors for Indexes	7-51
Advantages of Using <i>fillfactor</i>	7-52
Disadvantages of Using <i>fillfactor</i>	7-53
Using <i>sp_sysmon</i> to Observe the Effects of Changing <i>fillfactor</i>	7-53

8. Understanding the Query Optimizer

What Is Query Optimization?	8-1
Symptoms of Optimization Problems	8-1
Sources of Optimization Problems	8-2
Adaptive Server's Cost-Based Optimizer	8-2
Steps in Query Processing	8-3
Working with the Optimizer	8-4
How Is "Fast" Determined?	8-4
Query Optimization and Plans	8-5
Diagnostic Tools for Query Optimization	8-6
Using <i>showplan</i> and <i>noexec</i> Together	8-7
<i>noexec</i> and <i>statistics io</i>	8-7
Using <i>set statistics time</i>	8-7
Optimizer Strategies	8-8
Search Arguments and Using Indexes	8-9
SARGs in <i>where</i> and <i>having</i> Clauses	8-10
Indexable Search Argument Syntax	8-10
Search Argument Equivalents	8-11
Transitive Closure Applied to SARGs	8-12
Guidelines for Creating Search Arguments	8-12
Optimizing Joins	8-13
Join Syntax	8-14

How Joins Are Processed	8-14
Basic Join Processing	8-15
Choice of Inner and Outer Tables	8-15
Saving I/O Using the Reformatting Strategy	8-17
Index Density and Joins	8-18
Datatype Mismatches and Joins	8-19
Join Permutations	8-20
Joins in Queries with More Than Four Tables	8-20
Optimization of <i>or</i> clauses and <i>in (values_list)</i>	8-23
<i>or</i> syntax	8-23
<i>in (values_list)</i> Converts to <i>or</i> Processing	8-23
How <i>or</i> Clauses Are Processed	8-23
<i>or</i> Clauses and Table Scans	8-24
Multiple Matching Index Scans	8-24
The OR Strategy	8-24
Locking and the OR Strategy	8-25
Optimizing Aggregates	8-26
Combining <i>max</i> and <i>min</i> Aggregates	8-27
Optimizing Subqueries	8-28
Flattening <i>in</i> , <i>any</i> , and <i>exists</i> Subqueries	8-28
Flattening Expression Subqueries	8-29
Materializing Subquery Results	8-29
Noncorrelated Expression Subqueries	8-30
Quantified Predicate Subqueries Containing Aggregates	8-30
Short Circuiting	8-31
Subquery Introduced with an <i>and</i> Clause	8-31
Subquery Introduced with an <i>or</i> Clause	8-32
Subquery Results Caching	8-32
Displaying Subquery Cache Information	8-33
Optimizing Subqueries	8-33
Update Operations	8-34
Direct Updates	8-34
In-Place Updates	8-35
Cheap Direct Updates	8-36
Expensive Direct Updates	8-37
Deferred Updates	8-39
Deferred Index Inserts	8-39
Optimizing Updates	8-42
Effects of Update Types and Indexes on Update Modes	8-43
Choosing Fixed-Length Datatypes for Direct Updates	8-44
Using <i>max_rows_per_page</i> to Increase Direct Updates	8-44

Using <i>sp_sysmon</i> While Tuning Updates	8-45
---	------

9. Understanding Query Plans

Using <i>showplan</i>	9-1
Combining <i>showplan</i> and <i>noexec</i>	9-1
More Tips on Using <i>showplan</i>	9-2
Basic <i>showplan</i> Messages	9-2
Query Plan Delimiter Message	9-3
Step Message	9-3
Query Type Message	9-4
FROM TABLE Message	9-4
FROM TABLE and Referential Integrity	9-6
TO TABLE Message	9-7
Nested Iteration Message	9-8
Update Mode Messages	9-9
Direct Update Mode	9-9
Deferred Mode	9-10
Deferred Index and Deferred Varcol Messages	9-11
<i>showplan</i> Messages for Query Clauses	9-12
GROUP BY Message	9-13
Selecting into a Worktable	9-13
Grouped Aggregate Message	9-14
Grouped Aggregates and <i>group by</i>	9-15
<i>compute by</i> Message	9-16
Ungrouped Aggregate Message	9-17
Ungrouped Aggregates	9-17
<i>compute</i> Messages	9-18
Messages for <i>order by</i> and <i>distinct</i>	9-19
Worktable Message for <i>distinct</i>	9-19
Worktable Message for <i>order by</i>	9-21
Sorting Messages	9-22
This Step Involves Sorting Message	9-22
GETSORTED Message	9-22
Serial or Parallel Sort Message	9-22
<i>showplan</i> Messages Describing Access Methods, Caching, and I/O Cost	9-23
Auxiliary Scan Descriptors Message	9-24
Table Scan Message	9-26
Clustered Index Message	9-27
Index Name Message	9-27
Scan Direction Messages	9-28
Positioning Messages	9-28

Scanning Messages	9-29
Index Covering Message	9-30
Keys Message	9-31
Matching Index Scans Message	9-32
Dynamic Index Message	9-33
.	9-33
Reformatting Message	9-35
Trigger Log Scan Message	9-37
I/O Size Message	9-37
Cache Strategy Message	9-38
Total Estimated I/O Cost Message	9-38
<i>showplan</i> Messages for Parallel Queries	9-39
Executed in Parallel Messages	9-40
Coordinating Process Message	9-40
Worker Processes Message	9-41
Scan Type Message	9-41
Merge Messages	9-42
Run-Time Adjustment Message	9-45
<i>showplan</i> Messages for Subqueries	9-45
Output for Flattened or Materialized Subqueries	9-47
Flattened Queries	9-47
Materialized Queries	9-48
Structure of Subquery <i>showplan</i> Output	9-50
Subquery Execution Message	9-52
Nesting Level Delimiter Message	9-52
Subquery Plan Start Delimiter	9-52
Subquery Plan End Delimiter	9-52
Type of Subquery	9-52
Subquery Predicates	9-53
Internal Subquery Aggregates	9-53
Grouped or Ungrouped Messages	9-54
Quantified Predicate Subqueries and the ANY Aggregate	9-54
Expression Subqueries and the ONCE Aggregate	9-55
Subqueries with <i>distinct</i> and the ONCE-UNIQUE Aggregate	9-57
Existence Join Message	9-58
Subqueries That Perform Existence Tests	9-58

10. Advanced Optimizing Techniques

Why Special Optimizing Techniques May Be Needed	10-1
Specifying Optimizer Choices	10-2

Specifying Table Order in Joins	10-3
<i>forceplan</i> example	10-3
Risks of Using <i>forceplan</i>	10-7
Things to Try Before Using <i>forceplan</i>	10-7
Increasing the Number of Tables Considered by the Optimizer	10-8
Specifying an Index for a Query	10-8
Risks of Specifying Indexes in Queries	10-10
Things to Try Before Specifying Indexes	10-10
Specifying I/O Size in a Query	10-10
Index Type and Large I/O	10-12
When <i>prefetch</i> Specification Is Not Followed	10-12
<i>set prefetch on</i>	10-13
Specifying the Cache Strategy	10-13
Specifying Cache Strategy in <i>select</i> , <i>delete</i> , and <i>update</i> Statements	10-14
Controlling Large I/O and Cache Strategies	10-14
Getting Information on Cache Strategies	10-15
Suggesting a Degree of Parallelism for a Query	10-15
Query Level <i>parallel</i> Clause Examples	10-17
Tuning with <i>dbcc traceon 302</i>	10-17
Invoking the <i>dbcc</i> Trace Facility	10-18
General Tips for Tuning with <i>dbcc traceon(302)</i>	10-18
Checking for Join Columns and Search Arguments	10-18
Determining How the Optimizer Estimates I/O Costs	10-19
Trace Facility Output	10-19
Identifying the Table	10-20
Estimating the Table Size	10-20
Identifying the <i>where</i> Clause	10-21
Output for Range Queries	10-21
Specified Indexes	10-22
Calculating Base Cost	10-22
Costing Indexes	10-23
Index Statistics Used in <i>dbcc traceon(302)</i>	10-24
Evaluating Statistics for Search Clauses	10-24
Distribution Page Value Matches	10-25
Values Between Steps or Out of Range	10-25
Range Query Messages	10-26
Search Clauses with Unknown Values	10-26
Cost Estimates and Selectivity	10-28
Estimating Selectivity for Search Clauses	10-29
Estimating Selectivity for Join Clauses	10-29

11. Transact-SQL Performance Tips

“Greater Than” Queries	11-1
<i>not exists</i> Tests	11-2
Variables vs. Parameters in <i>where</i> Clauses	11-3
Count vs. Exists	11-4
<i>or</i> Clauses vs. Unions in Joins	11-5
Aggregates	11-5
Joins and Datatypes	11-6
Null vs. Not Null Character and Binary Columns.	11-7
Forcing the Conversion to the Other Side of the Join	11-7
Parameters and Datatypes.	11-8

12. Cursors and Performance

What Is a Cursor?	12-1
Set-Oriented vs. Row-Oriented Programming.	12-2
Cursors: A Simple Example	12-3
Resources Required at Each Stage	12-4
Memory Use and Execute Cursors.	12-5
Cursor Modes: Read-Only and Update.	12-6
Read-Only vs. Update	12-6
Index Use and Requirements for Cursors	12-6
Comparing Performance With and Without Cursors	12-7
Sample Stored Procedure Without a Cursor.	12-7
Sample Stored Procedure With a Cursor.	12-8
Cursor vs. Non-Cursor Performance Comparison	12-10
Cursor vs. Non-Cursor Performance Explanation.	12-10
Locking with Read-Only Cursors	12-10
Locking with Update Cursors	12-11
Update Cursors: Experiment Results.	12-12
Isolation Levels and Cursors.	12-13
Partitioned Heap Tables and Cursors.	12-13
Optimizing Tips for Cursors	12-14
Optimizing for Cursor Selects Using a Cursor.	12-14
Using <i>union</i> Instead of <i>or</i> Clauses or <i>in</i> Lists	12-14
Declaring the Cursor’s Intent	12-15
Specifying Column Names in the <i>for update</i> Clause	12-15
Using <i>set cursor rows</i>	12-16
Keeping Cursors Open Across Commits and Rollbacks.	12-16
Opening Multiple Cursors on a Single Connection.	12-17

13. Introduction to Parallel Query Processing

Types of Queries That Can Benefit from Parallel Processing	13-2
Adaptive Server's Worker Process Model	13-3
Parallel Query Execution	13-4
Returning Results from Parallel Queries	13-5
Types of Parallel Data Access	13-6
Hash-Based Table Scans	13-7
Partition-Based Scans	13-8
Hash-Based Nonclustered Index Scans	13-8
Parallel Processing for Two Tables in a Join	13-9
<i>showplan</i> Messages for Parallel Queries	13-10
Controlling the Degree of Parallelism	13-11
Configuration Parameters for Controlling Parallelism	13-12
How Limits Apply to Query Plans	13-12
How the Limits Work In Combination	13-13
Examples of Setting Parallel Configuration Parameters	13-14
Using <i>set</i> Options to Control Parallelism for a Session	13-14
<i>set</i> Command Examples	13-15
Controlling Parallelism for a Query	13-15
Query Level <i>parallel</i> Clause Examples	13-15
Worker Process Availability and Query Execution	13-16
Other Configuration Parameters for Parallel Processing	13-16
Commands for Working with Partitioned Tables	13-17
Balancing Resources and Performance	13-18
CPU Resources	13-19
Disk Resources and I/O	13-19
Tuning Example: CPU and I/O Saturation	13-19
Guidelines for Parallel Query Configuration	13-20
Hardware Guidelines	13-20
Working with Your Performance Goals and Hardware Guidelines ..	13-21
Examples of Parallel Query Tuning	13-22
Improving the Performance of a Table Scan	13-22
Improving the Performance of a Nonclustered Index Scan	13-23
Guidelines for Partitioning and Parallel Degree	13-23
Experimenting with Data Subsets	13-24
System Level Impacts	13-25
Locking Issues	13-25
Device Issues	13-25
Procedure Cache Effects	13-26
When Parallel Query Results Can Differ	13-26
Queries That Use <i>set rowcount</i>	13-27

Queries That Set Local Variables	13-27
Achieving Consistent Results	13-27

14. Parallel Query Optimization

What Is Parallel Query Optimization?	14-1
Optimizing for Response Time vs. Total Work	14-2
When Is Parallel Query Optimization Performed?	14-3
Overhead Cost of Parallel Queries	14-3
Factors That Are Not Considered	14-4
Parallel Access Methods	14-4
Parallel Partition Scan	14-5
Requirements for Consideration	14-5
Cost Model	14-6
Parallel Hash-Based Table Scan	14-6
Requirements for Consideration	14-7
Cost Model	14-7
Parallel Clustered Index Partition Scan	14-7
Requirements for Consideration	14-8
Cost Model	14-8
Parallel Nonclustered Index Hash-Based Scan	14-9
Cost Model	14-10
Additional Parallel Strategies	14-11
Partitioned Worktables	14-11
Parallel Sorting	14-11
Summary of Parallel Access Methods	14-11
Selecting Parallel Access Methods	14-12
Degree of Parallelism for Parallel Queries	14-13
Upper Limit to Degree of Parallelism	14-14
Optimized Degree of Parallelism	14-14
Degree of Parallelism Examples	14-15
Run-Time Adjustments to Worker Processes	14-16
Parallel Strategies for Joins	14-17
Join Order and Cost	14-17
Determining the Cost of Serial Join Orders	14-17
Determining the Cost of Parallel Join Orders	14-17
Degree of Parallelism for Joins	14-18
Alternative Plans	14-20
Computing the Degree of Parallelism for Joins	14-20
Outer Joins and the Existence Clause	14-21
Parallel Query Examples	14-21
Single-Table Scans	14-21

Table Partition Scan Example	14-22
Multitable Joins	14-23
Parallel Join Optimization and Join Orders	14-24
Subqueries	14-27
Example of Subqueries in Parallel Queries	14-27
Queries That Require Worktables	14-28
union Queries	14-28
Queries with Aggregates	14-29
select into Statements	14-29
Parallel select into Example	14-30
Run-Time Adjustment of Worker Processes	14-30
How Adaptive Server Adjusts a Query Plan	14-31
Evaluating the Effect of Run-Time Adjustments	14-32
Recognizing and Managing Run-Time Adjustments	14-32
Using set process_limit_action	14-32
Using showplan	14-33
Reducing the Likelihood of Run-Time Adjustments	14-36
Checking Run-Time Adjustments with sp_sysmon	14-36
Diagnosing Parallel Performance Problems	14-37
Query Does Not Run in Parallel (When You Think It Should)	14-37
Parallel Performance Is Not As Good As Expected	14-38
Calling Technical Support for Diagnosis	14-39
Resource Limits for Parallel Queries	14-39

15. Parallel Sorting

Commands That Benefit from Parallel Sorting	15-1
Parallel Sort Requirements and Resources Overview	15-2
Overview of the Parallel Sorting Strategy	15-3
Sampling and Creating a Distribution Map	15-4
Dynamic Range Partitioning	15-5
Range Sorting	15-5
Merging Results	15-6
Configuring Resources for Parallel Sorting	15-6
Worker Process Requirements During Parallel Sorts	15-6
Worker Process Requirements for Creating Indexes	15-7
Using with consumers While Creating Indexes	15-9
Worker Process Requirements for select Query Sorts	15-10
Caches, Sort Buffers, and Parallel Sorts	15-10
Cache Bindings	15-11
How the Number of Sort Buffers Affects Sort Performance	15-11

Sort Buffer Configuration Guidelines	15-11
Using Less Than the Configured Number of Sort Buffers	15-12
Configuring the <i>number of sort buffers</i> Parameter	15-13
Procedure for Estimating Merge Runs and I/O	15-14
Configuring Caches for Large I/O During Parallel Sorting	15-15
Balancing Sort Buffers and Large I/O Configuration	15-16
Disk Requirements	15-16
Space Requirements for Creating Indexes	15-17
Space Requirements for Worktable Sorts	15-17
Number of Devices in the Target Segment	15-17
Recovery Considerations	15-18
Tools for Observing and Tuning Sort Behavior	15-18
Using <i>set sort_resources on</i>	15-19
Sort Examples	15-20
Using <i>sp_sysmon</i> to Tune Index Creation	15-23

16. Memory Use and Performance

How Memory Affects Performance	16-1
How Much Memory to Configure	16-2
Caches in Adaptive Server	16-3
The Procedure Cache	16-4
Getting Information About the Procedure Cache Size	16-5
proc buffers	16-5
proc headers	16-6
Procedure Cache Sizing	16-6
Estimating Stored Procedure Size	16-6
Monitoring Procedure Cache Performance	16-7
Procedure Cache Errors	16-7
The Data Cache	16-7
Default Cache at Installation Time	16-7
Page Aging in Data Cache	16-8
Effect of Data Cache on Retrievals	16-8
Effect of Data Modifications on the Cache	16-9
Data Cache Performance	16-10
Testing Data Cache Performance	16-10
Cache Hit Ratio for a Single Query	16-11
Cache Hit Ratio Information from <i>sp_sysmon</i>	16-11
Named Data Caches	16-12
Named Data Caches and Performance	16-12
Large I/Os and Performance	16-13

Types of Queries That Can Benefit from Large I/O	16-14
The Optimizer and Cache Choices	16-16
Choosing the Right Mix of I/O Sizes for a Cache	16-16
Commands to Configure Named Data Caches	16-17
Commands for Tuning Query I/O Strategies and Sizes	16-18
Cache Replacement Strategies and Policies	16-18
Cache Replacement Strategies	16-18
Cache Replacement Policies	16-19
Cache Replacement Policy Concepts	16-19
Comparing Cache Replacement Policies	16-22
How Relaxed LRU Caches Work	16-23
Configuring and Tuning Named Caches	16-23
Cache Configuration Goals	16-24
Development vs. Production Systems	16-25
Gather Data, Plan, and Then Implement	16-25
Evaluating Cache Needs	16-26
Named Data Cache Recommendations	16-27
Sizing Caches for Special Objects, <i>tempdb</i> , and Transaction Logs	16-28
Determining Cache Sizes for Special Tables or Indexes	16-28
Examining Cache Needs for <i>tempdb</i>	16-28
Examining Cache Needs for Transaction Logs	16-29
Choosing the I/O Size for the Transaction Log	16-29
Configuring for Large Log I/O Size	16-30
Additional Tuning Tips for Log Caches	16-31
Basing Data Pool Sizes on Query Plans and I/O	16-31
Checking I/O Size for Queries	16-32
Configuring Buffer Wash Size	16-33
Choosing Caches for Relaxed LRU Replacement Policy	16-33
Configuring Relaxed LRU Replacement for Database Logs	16-34
Relaxed LRU Replacement for Lookup Tables and Indexes	16-34
Overhead of Pool Configuration and Binding Objects	16-34
Pool Configuration Overhead	16-34
Cache Binding Overhead	16-34
Maintaining Data Cache Performance for Large I/O	16-35
Diagnosing Excessive I/O Counts	16-36
Using <i>sp_sysmon</i> to Check Large I/O Performance	16-38
Re-Creating Indexes to Eliminate Fragmentation	16-38
Using Fillfactor for Data Cache Performance	16-39
Speed of Recovery	16-39
Tuning the Recovery Interval	16-40
Effects of the Housekeeper Task on Recovery Time	16-40

Auditing and Performance	16-41
Sizing the Audit Queue	16-41
Auditing Performance Guidelines	16-42

17. Controlling Physical Data Placement

How Object Placement Can Improve Performance	17-1
Symptoms of Poor Object Placement	17-2
Underlying Problems	17-3
Using <i>sp_sysmon</i> While Changing Data Placement	17-3
Terminology and Concepts	17-3
Guidelines for Improving I/O Performance	17-4
Spreading Data Across Disks to Avoid I/O Contention	17-5
Avoiding Physical Contention in Parallel Join Queries	17-5
Isolating Server-Wide I/O from Database I/O	17-6
Where to Place <i>tempdb</i>	17-6
Where to Place <i>sybsecurity</i>	17-7
Keeping Transaction Logs on a Separate Disk	17-7
Mirroring a Device on a Separate Disk	17-9
Device Mirroring Performance Issues	17-9
Why Use Serial Mode?	17-10
Creating Objects on Segments	17-11
Why Use Segments?	17-12
Separating Tables and Indexes	17-12
Splitting a Large Table Across Devices	17-13
Moving Text Storage to a Separate Device	17-13
Partitioning Tables for Performance	17-14
User Transparency	17-14
Partitioned Tables and Parallel Query Processing	17-15
Distributing Data Across Partitions	17-15
Improving Insert Performance with Partitions	17-16
Page Contention During Inserts	17-16
How Partitions Address Page Contention	17-16
Selecting Heap Tables to Partition	17-17
Restrictions on Partitioned Tables	17-17
Partition-Related Configuration Parameters	17-17
How Adaptive Server Distributes Partitions on Devices	17-18
Effects of Partitioning on System Tables	17-19
Space Planning for Partitioned Tables	17-20
Read-Only Tables	17-20
Read-Mostly Tables	17-21
Tables with Random Data Modification	17-21

Commands for Partitioning Tables	17-22
<i>alter table...partition</i> Syntax	17-22
<i>alter table...unpartition</i> Syntax	17-23
Changing the Number of Partitions	17-23
Distributing Data Evenly Across Partitions	17-24
Commands to Create and Drop Clustered Indexes	17-24
Using <i>drop index</i> and <i>create clustered index</i>	17-25
Using Constraints and <i>alter table</i>	17-25
Special Concerns for Partitioned Tables and Clustered Indexes	17-25
Using Parallel <i>bcp</i> to Copy Data into Partitions	17-26
Parallel Copy and Locks	17-26
Getting Information About Partitions	17-27
Checking Data Distribution on Devices with <i>sp_helpsegment</i>	17-28
Effects of Imbalance of Data on Segments and Partitions	17-29
Determining the Number of Pages in a Partition	17-29
Updating Partition Statistics	17-30
Syntax for <i>update partition statistics</i>	17-31
Steps for Partitioning Tables	17-31
Backing Up the Database After Partitioning Tables	17-31
The Table Does Not Exist	17-32
The Table Exists Elsewhere in the Database	17-33
The Table Exists on the Segment	17-33
Redistributing Data	17-34
Adding Devices to a Segment	17-37
Special Procedures for Difficult Situations	17-38
A Technique for Clustered Indexes on Large Tables	17-38
A Complex Alternative for Clustered Indexes	17-39
Problems When Devices for Partitioned Tables Are Full	17-41
Adding Disks When Devices Are Full	17-41
Adding Disks When Devices Are Nearly Full	17-42
Maintenance Issues and Partitioned Tables	17-43
Regular Maintenance Checks for Partitioned Tables	17-44

18. Tuning Asynchronous Prefetch

How Asynchronous Prefetch Improves Performance	18-1
Improving Query Performance by Prefetching Pages	18-2
Prefetching Control Mechanisms in a Multiuser Environment	18-3
The Look-Ahead Set During Recovery	18-3
Prefetching Log Pages	18-3
Prefetching Data and Index Pages	18-4
The Look-Ahead Set During Sequential Scans	18-4

The Look-Ahead Set During Nonclustered Index Access	18-4
The Look-Ahead Set During <i>dbcc</i> Checks	18-5
Allocation Checking	18-5
<i>checkdb</i> and <i>checktable</i>	18-5
Look-Ahead Set Minimum and Maximum Sizes	18-5
When Prefetch Is Automatically Disabled	18-7
Flooding Pools	18-7
I/O System Overloads	18-7
Unnecessary Reads	18-8
Page Chain Fragmentation	18-8
Tuning Goals for Asynchronous Prefetch	18-10
Commands to Configure Asynchronous Prefetch	18-10
Asynchronous Prefetch and Other Performance Features	18-11
Large I/O and Asynchronous Prefetch	18-11
Sizing and Limits for the 16K Pool	18-11
Limits for the 2K Pool	18-12
Fetch-and-Discard (MRU) Scans and Asynchronous Prefetch	18-12
Parallel Scans, Large I/Os, and Asynchronous Prefetch	18-12
Hash-Based Scans	18-13
Partition Scans	18-13
Special Settings for Asynchronous Prefetch Limits	18-14
Setting Asynchronous Prefetch Limits for Recovery	18-14
Setting Asynchronous Prefetch Limits for <i>dbcc</i>	18-14
Maintenance Activities for High Prefetch Performance	18-15
Eliminating Kinks in Heap Tables	18-15
Eliminating Kinks in Clustered Index Tables	18-15
Eliminating Kinks in Nonclustered Indexes	18-15
Performance Monitoring and Asynchronous Prefetch	18-15

19. *tempdb* Performance Issues

How <i>tempdb</i> Affects Performance	19-1
Main Solution Areas for <i>tempdb</i> Performance	19-2
Types and Uses of Temporary Tables	19-2
Truly Temporary Tables	19-2
Regular User Tables	19-3
Worktables	19-3
Initial Allocation of <i>tempdb</i>	19-4
Sizing <i>tempdb</i>	19-4
Information for Sizing <i>tempdb</i>	19-5
Estimating Table Sizes in <i>tempdb</i>	19-5

<i>tempdb</i> Sizing Formula	19-6
Example of <i>tempdb</i> Sizing	19-7
Placing <i>tempdb</i>	19-8
Dropping the master Device from <i>tempdb</i> Segments	19-8
Using Multiple Disks for Parallel Query Performance	19-10
Binding <i>tempdb</i> to Its Own Cache	19-10
Commands for Cache Binding	19-10
Temporary Tables and Locking	19-10
Minimizing Logging in <i>tempdb</i>	19-11
Minimizing Logging with <i>select into</i>	19-11
Minimizing Logging by Using Shorter Rows	19-11
Optimizing Temporary Tables	19-12
Creating Indexes on Temporary Tables	19-13
Breaking <i>tempdb</i> Uses into Multiple Procedures	19-13
Creating Nested Procedures with Temporary Tables	19-14

20. Networks and Performance

Why Study the Network?	20-1
Potential Network-Based Performance Problems	20-1
Basic Questions About Networks and Performance	20-2
Techniques Summary	20-2
Using <i>sp_sysmon</i> While Changing Network Configuration	20-3
How Adaptive Server Uses the Network	20-3
Changing Network Packet Sizes	20-3
Large vs. Default Packet Sizes for User Connections	20-4
Number of Packets Is Important	20-4
Point of Diminishing Returns	20-5
Client Commands for Larger Packet Sizes	20-5
Evaluation Tools with Adaptive Server	20-6
Evaluation Tools Outside of Adaptive Server	20-6
Techniques for Reducing Network Traffic	20-7
Server-Based Techniques for Reducing Traffic	20-7
Using Stored Procedures to Reduce Network Traffic	20-7
Ask for Only the Information You Need	20-7
Fill Up Packets When Using Cursors	20-7
Large Transfers	20-8
Network Overload	20-8
Impact of Other Server Activities	20-8
Login Protocol	20-9
Single User vs. Multiple Users	20-9

Guidelines for Improving Network Performance	20-10
Choose the Right Packet Size for the Task	20-10
Isolate Heavy Network Users	20-12
Set <i>tcp no delay</i> on TCP Networks	20-12
Configure Multiple Network Listeners	20-12

21. How Adaptive Server Uses Engines and CPUs

Background Concepts	21-1
How Adaptive Server Processes Client Requests	21-2
Client Task Implementation	21-3
The Single-CPU Process Model	21-4
Scheduling Engines to the CPU	21-4
Scheduling Tasks to the Engine	21-6
Adaptive Server Execution Task Scheduling	21-7
Scheduling Client Task Processing Time	21-7
Guarding CPU Availability During Idle Time	21-9
Tuning Scheduling Parameters	21-10
The Adaptive Server SMP Process Model	21-10
Scheduling Engines to CPUs	21-11
Scheduling Adaptive Server Tasks to Engines	21-11
Multiple Network Engines	21-12
Multiple Run Queues	21-13
A Processing Scenario	21-14
Logging In and Assigning a Network Engine	21-16
Checking for Client Requests	21-16
Fulfilling a Client Request	21-16
Performing Disk I/O	21-16
Performing Network I/O	21-17
How the Housekeeper Task Improves CPU Utilization	21-17
Side Effects of the Housekeeper Task	21-17
Configuring the Housekeeper Task	21-18
Changing the Percentage by Which Writes Can Be Increased.	21-18
Disabling the Housekeeper Task	21-18
Allowing the Housekeeper Task to Work Continuously	21-18
Measuring CPU Usage	21-19
Single CPU Machines	21-19
Using <i>sp_monitor</i> to Measure CPU Usage	21-19
Using <i>sp_sysmon</i> to Measure CPU Usage	21-20
Operating System Commands and CPU Usage	21-20
Multiple CPU Machines	21-20
Determining When to Configure Additional Engines	21-20

Enabling Engine-to-CPU Affinity	21-21
Multiprocessor Application Design Guidelines	21-23
Multiple Indexes	21-23
Managing Disks	21-24
Adjusting the <i>fillfactor</i> for <i>create index</i> Commands	21-24
Setting <i>max_rows_per_page</i>	21-24
Transaction Length	21-24
Temporary Tables	21-25

22. Distributing Engine Resources Between Tasks

Using Execution Attributes to Manage Preferred Access to Resources	22-1
Types of Execution Classes	22-2
Predefined Execution Classes	22-2
User-Defined Execution Classes	22-3
Execution Class Attributes	22-3
Base Priority	22-4
Time Slice	22-5
Task-to-Engine Affinity	22-5
Setting Execution Class Attributes	22-7
Assigning Execution Classes	22-7
Engine Groups and Establishing Task-to-Engine Affinity	22-8
How Execution Class Bindings Affect Scheduling	22-10
Execution Class Bindings	22-10
Effect on Scheduling	22-10
Setting Attributes for a Session Only	22-12
Getting Information About Execution Class Bindings and Attributes	22-12
Rules for Determining Precedence and Scope	22-12
Multiple Execution Objects and ECs, Different Scopes	22-13
The Precedence Rule	22-13
The Scope Rule	22-14
Resolving a Precedence Conflict	22-15
Examples: Determining Precedence	22-16
Example Scenario Using Precedence Rules	22-18
Planning	22-19
Configuration	22-20
Execution Characteristics	22-20
Considerations for Engine Resource Distribution	22-21
Client Applications: OLTP and DSS	22-21
Unintrusive Client Applications	22-22
I/O-Bound Client Applications	22-22

Highly Critical Applications	22-22
Adaptive Server Logins: High Priority Users.....	22-22
Stored Procedures: “Hot Spots”	22-22
Algorithm for Successfully Distributing Engine Resources	22-23
Algorithm Guidelines	22-25
Environment Analysis and Planning.....	22-26
Analyzing the Environment	22-26
Example: Phase 1 – Analyzing Execution Object Behavior	22-27
Example: Phase 2 – Analyzing the Environment As a Whole	22-28
Performing Benchmark Tests	22-28
Setting Goals	22-29
Results Analysis and Tuning.....	22-29
Monitoring the Environment Over Time	22-29

23. Maintenance Activities and Performance

Creating or Altering a Database.....	23-1
Creating Indexes	23-3
Configuring Adaptive Server to Speed Sorting	23-3
Dumping the Database After Creating an Index	23-4
Creating an Index on Sorted Data	23-4
Backup and Recovery	23-5
Local Backups	23-6
Remote Backups	23-6
Online Backups.....	23-6
Using Thresholds to Prevent Running Out of Log Space	23-6
Minimizing Recovery Time.....	23-6
Recovery Order.....	23-7
Bulk Copy	23-7
Parallel Bulk Copy	23-7
Batches and Bulk Copy	23-8
Slow Bulk Copy	23-8
Improving Bulk Copy Performance.....	23-8
Replacing the Data in a Large Table.....	23-8
Adding Large Amounts of Data to a Table	23-9
Using Partitions and Multiple Bulk Copy Processes.....	23-9
Impacts on Other Users	23-9
Database Consistency Checker.....	23-9
Using <i>dbcc tune (cleanup)</i>	23-10

24. Monitoring Performance with *sp_sysmon*

Using <i>sp_sysmon</i>	24-2
When to Run <i>sp_sysmon</i>	24-3
Invoking <i>sp_sysmon</i>	24-4
Running <i>sp_sysmon</i> for a Fixed Time Interval	24-5
Running <i>sp_sysmon</i> Using <i>begin_sample</i> and <i>end_sample</i>	24-5
Specifying Report Sections for <i>sp_sysmon</i> Output	24-6
Specifying the Application Detail Parameter	24-6
Redirecting <i>sp_sysmon</i> Output to a File	24-7
How to Use <i>sp_sysmon</i> Reports	24-7
Reading <i>sp_sysmon</i> Output	24-8
Rows	24-9
Columns	24-9
Interpreting <i>sp_sysmon</i> Data	24-10
Per Second and Per Transaction Data	24-10
Percent of Total and Count Data	24-10
Per Engine Data	24-10
Total or Summary Data	24-11
Sample Interval and Time Reporting	24-11
Kernel Utilization	24-11
Sample Output for Kernel Utilization	24-12
Engine Busy Utilization	24-12
CPU Yields by Engine	24-14
Network Checks	24-14
Non-Blocking	24-15
Blocking	24-15
Total Network I/O Checks	24-15
Average Network I/Os per Check	24-16
Disk I/O Checks	24-16
Total Disk I/O Checks	24-16
Checks Returning I/O	24-16
Average Disk I/Os Returned	24-17
Worker Process Management	24-17
Sample Output for Worker Process Management	24-18
Worker Process Requests	24-18
Worker Process Usage	24-18
Memory Requests for Worker Processes	24-19
Avg Mem Ever Used by a WP	24-19
Parallel Query Management	24-20
Sample Output for Parallel Query Management	24-20
Parallel Query Usage	24-21

Merge Lock Requests	24-21
Sort Buffer Waits	24-22
Task Management	24-22
Sample Output for Task Management	24-22
Connections Opened	24-23
Task Context Switches by Engine	24-23
Task Context Switches Due To	24-23
Voluntary Yields	24-24
Cache Search Misses	24-24
System Disk Writes	24-25
I/O Pacing	24-25
Logical Lock Contention	24-25
Address Lock Contention	24-26
Log Semaphore Contention	24-26
Group Commit Sleeps	24-27
Last Log Page Writes	24-27
Modify Conflicts	24-28
I/O Device Contention	24-28
Network Packet Received	24-28
Network Packet Sent	24-29
SYSINDEXES Lookup	24-29
Other Causes	24-29
Application Management	24-30
Requesting Detailed Application Information	24-30
Sample Output for Application Management	24-30
Application Statistics Summary (All Applications)	24-32
Priority Changes	24-32
Allotted Slices Exhausted	24-33
Skipped Tasks By Engine	24-34
Engine Scope Changes	24-34
Application Statistics per Application or per Application and Login	24-34
Application Activity	24-34
Application Priority Changes	24-35
Application I/Os Completed	24-35
Resource Limits Violated	24-36
ESP Management	24-36
Sample Output for ESP Management	24-36
ESP Requests	24-36
Avg. Time to Execute an ESP	24-36
Monitor Access to Executing SQL	24-36
Sample Output for Monitor Access to Executing SQL	24-37

Waits on Execution Plans	24-37
Number of SQL Text Overflows	24-37
Maximum SQL Text Requested	24-37
Transaction Profile	24-37
Sample Output for Transaction Profile	24-38
Transaction Summary	24-38
Committed Transactions	24-38
Transaction Detail	24-40
Inserts	24-40
Updates	24-42
Deletes	24-42
Transaction Management	24-43
Sample Output for Transaction Management	24-43
ULC Flushes to Transaction Log	24-44
By Full ULC	24-45
By End Transaction	24-45
By Change of Database	24-45
By System Log Record and By Other	24-45
Total ULC Flushes	24-45
ULC Log Records	24-46
Maximum ULC Size	24-46
ULC Semaphore Requests	24-46
Log Semaphore Requests	24-47
Log Semaphore Contention and User Log Caches	24-47
Transaction Log Writes	24-48
Transaction Log Allocations	24-48
Avg # Writes per Log Page	24-48
Index Management	24-49
Sample Output for Index Management	24-49
Nonclustered Maintenance	24-50
Inserts and Updates Requiring Maintenance to Indexes	24-50
Deletes Requiring Maintenance	24-51
Row ID Updates from Clustered Split	24-51
Page Splits	24-52
Reducing Page Splits for Ascending Key Inserts	24-52
Default Data Page Splitting	24-53
Effects of Ascending Inserts	24-54
Setting Ascending Inserts Mode for a Table	24-55
Retries and Deadlocks	24-55
Empty Page Flushes	24-56
Add Index Level	24-56

Page Shrinks	24-56
Index Scans	24-57
Metadata Cache Management	24-57
Sample Output for Metadata Cache Management	24-57
Open Object, Index, and Database Usage	24-58
Open Object and Open Index Spinlock Contention	24-59
Open Index Hash Spinlock Contention	24-59
Lock Management	24-59
Sample Output for Lock Management	24-59
Lock Summary	24-62
Total Lock Requests	24-62
Average Lock Contention	24-62
Deadlock Percentage	24-62
Lock Detail	24-62
Address Locks	24-63
Last Page Locks on Heaps	24-63
Deadlocks by Lock Type	24-64
Deadlock Detection	24-64
Deadlock Searches	24-65
Searches Skipped	24-65
Average Deadlocks per Search	24-65
Lock Promotions	24-66
Data Cache Management	24-66
Sample Output for Data Cache Management	24-68
Cache Statistics Summary (All Caches)	24-70
Cache Search Summary	24-70
Cache Turnover	24-71
Cache Strategy Summary	24-71
Large I/O Usage	24-71
Large I/O Effectiveness	24-72
Asynchronous Prefetch Activity Report	24-73
Other Asynchronous Prefetch Statistics	24-74
Dirty Read Behavior	24-75
Cache Management By Cache	24-76
Spinlock Contention	24-76
Utilization	24-76
Cache Search, Hit, and Miss Information	24-77
Pool Turnover	24-78
Buffer Wash Behavior	24-80
Cache Strategy	24-81
Large I/O Usage	24-81

Large I/O Detail	24-83
Dirty Read Behavior	24-83
Procedure Cache Management	24-83
Sample Output for Procedure Cache Management	24-83
Procedure Requests	24-84
Procedure Reads from Disk	24-84
Procedure Writes to Disk	24-84
Procedure Removals	24-84
Memory Management	24-85
Sample Output for Memory Management	24-85
Pages Allocated	24-85
Pages Released	24-85
Recovery Management	24-85
Sample Output for Recovery Management	24-85
Checkpoints	24-86
Number of Normal Checkpoints	24-86
Number of Free Checkpoints	24-87
Total Checkpoints	24-87
Average Time per Normal Checkpoint	24-87
Average Time per Free Checkpoint	24-87
Increasing the Housekeeper Batch Limit	24-87
Disk I/O Management	24-88
Sample Output for Disk I/O Management	24-89
Maximum Outstanding I/Os	24-90
I/Os Delayed By	24-91
Disk I/O Structures	24-91
Server Configuration Limit	24-91
Engine Configuration Limit	24-91
Operating System Limit	24-91
Requested and Completed Disk I/Os	24-92
Total Requested Disk I/Os	24-92
Completed Disk I/Os	24-92
Device Activity Detail	24-93
Reads and Writes	24-93
Total I/Os	24-93
Device Semaphore Granted and Waited	24-94
Network I/O Management	24-94
Sample Output for Network I/O Management	24-95
Total Network I/Os Requests	24-96
Network I/Os Delayed	24-97
Total TDS Packets Received	24-97

Total Bytes Received.....	24-97
Average Bytes Received per Packet.....	24-97
Total TDS Packets Sent.....	24-97
Total Bytes Sent.....	24-97
Average Bytes Sent per Packet.....	24-97
Reducing Packet Overhead.....	24-98

Index

List of Figures

Figure 1-1:	Adaptive Server system model.....	1-2
Figure 2-1:	Database design	2-2
Figure 2-2:	Levels of normalization	2-3
Figure 2-3:	A table that violates first normal form	2-5
Figure 2-4:	Correcting first normal form violations by creating two tables.....	2-5
Figure 2-5:	A table that violates second normal form.....	2-6
Figure 2-6:	Correcting second normal form violations by creating two tables	2-6
Figure 2-7:	A table that violates Third Normal Form.....	2-7
Figure 2-8:	Correcting Third Normal Form violations by creating two tables.....	2-7
Figure 2-9:	Balancing denormalization issues.....	2-9
Figure 2-10:	Denormalizing by adding redundant columns.....	2-11
Figure 2-11:	Denormalizing by adding derived columns.....	2-12
Figure 2-12:	Denormalizing by collapsing tables.....	2-13
Figure 2-13:	Denormalizing by duplicating tables	2-13
Figure 2-14:	Horizontal and vertical partitioning of tables.....	2-14
Figure 2-15:	Horizontal partitioning of active and inactive data	2-15
Figure 2-16:	Vertically partitioning a table.....	2-16
Figure 2-17:	Using triggers to maintain normalized data.....	2-17
Figure 2-18:	Maintaining denormalized data via application logic	2-17
Figure 2-19:	Using batch reconciliation to maintain data.....	2-18
Figure 3-1:	An Adaptive Server data page	3-3
Figure 3-2:	Row density	3-4
Figure 3-3:	Page linkage.....	3-6
Figure 3-4:	Text and image data storage.....	3-7
Figure 3-5:	OAM page and allocation page pointers	3-10
Figure 3-6:	Selecting from a heap	3-12
Figure 3-7:	Inserting a row into a heap table	3-13
Figure 3-8:	Deleting rows from a heap table.....	3-14
Figure 3-9:	Strict LRU strategy takes a clean page from the LRU end of the cache.....	3-17
Figure 3-10:	Relaxed LRU cache policy	3-17
Figure 3-11:	MRU strategy places pages just before the wash marker	3-18
Figure 3-12:	Finding a needed page in cache	3-18
Figure 3-13:	Inserts to a heap page in the data cache	3-20
Figure 3-14:	Data vs. log I/O	3-23
Figure 4-1:	A simplified index schematic.....	4-2
Figure 4-2:	Index levels and page chains	4-4
Figure 4-3:	Clustered index on last name.....	4-5
Figure 4-4:	Selecting a row using a clustered index	4-6

Figure 4-5:	Inserting a row into a table with a clustered index	4-7
Figure 4-6:	Page splitting in a table with a clustered index.....	4-8
Figure 4-7:	Adding an overflow page to a nonunique clustered index.....	4-10
Figure 4-8:	Deleting a row from a table with a clustered index.....	4-11
Figure 4-9:	Deleting the last row on a page (before the delete)	4-12
Figure 4-10:	Deleting the last row on a page (after the delete)	4-13
Figure 4-11:	Data page with the offset table	4-15
Figure 4-12:	Row offset table after an insert	4-15
Figure 4-13:	Nonclustered index structure	4-17
Figure 4-14:	Selecting rows using a nonclustered index.....	4-18
Figure 4-15:	An insert with a nonclustered index.....	4-19
Figure 4-16:	Deleting a row from a table with a nonclustered index.....	4-20
Figure 4-17:	Matching index access does not have to read the data row	4-22
Figure 4-18:	A nonmatching index scan.....	4-23
Figure 4-19:	Caching used for a point query via a nonclustered index.....	4-24
Figure 4-20:	Finding the root index page in cache.....	4-25
Figure 4-21:	Caching with separate caches for data and log.....	4-25
Figure 4-22:	Index page recycling in the cache.....	4-26
Figure 5-1:	Consistency levels in transactions.....	5-2
Figure 5-2:	Demand locking with serial query execution.....	5-9
Figure 5-3:	Demand locking with parallel query execution.....	5-10
Figure 5-4:	Dirty reads in transactions	5-13
Figure 5-5:	Transaction isolation level 1 prevents dirty reads	5-15
Figure 5-6:	Nonrepeatable reads in transactions.....	5-16
Figure 5-7:	Phantoms in transactions	5-17
Figure 5-8:	Avoiding phantoms in transactions	5-18
Figure 5-9:	Locking example between two transactions.....	5-24
Figure 5-10:	Deadlocks in transactions.....	5-29
Figure 5-11:	A deadlock between two processes	5-29
Figure 5-12:	A deadlock involving a family of worker processes	5-44
Figure 5-13:	Lock promotion logic	5-49
Figure 7-1:	Query processing analysis tools and query processing	7-9
Figure 7-2:	Formula for computing table scan time	7-19
Figure 7-3:	Computing reads for a clustered index range query.....	7-20
Figure 7-4:	Range query on a clustered index	7-21
Figure 7-5:	Range query with a covering nonclustered index	7-22
Figure 7-6:	Computing reads for a covering nonclustered index range query	7-23
Figure 7-7:	Computing reads for a nonclustered index range query.....	7-24
Figure 7-8:	An order by query using a clustered index	7-26
Figure 7-9:	An order by desc query using a clustered index.....	7-27
Figure 7-10:	Sample rows for small and large index entries	7-38

Figure 7-11:	Formulas for computing number of distribution page values	7-43
Figure 7-12:	Building the distribution page.....	7-44
Figure 7-13:	Table and clustered index with fillfactor set to 50 percent	7-52
Figure 8-1:	Query execution steps.....	8-3
Figure 8-2:	Formula for converting ticks to milliseconds	8-8
Figure 8-3:	SARGs and index choices	8-13
Figure 8-4:	Nesting of tables during a join.....	8-15
Figure 8-5:	Alternate join orders and page reads.....	8-16
Figure 8-6:	Resolving or queries.....	8-25
Figure 8-7:	In-place update	8-35
Figure 8-8:	Cheap direct update	8-37
Figure 8-9:	Expensive direct update	8-38
Figure 8-10:	Deferred index update	8-41
Figure 9-1:	Subquery showplan output structure.....	9-51
Figure 10-1:	Extreme negative effects of using forceplan	10-7
Figure 12-1:	Cursor example.....	12-1
Figure 12-2:	Cursor flowchart.....	12-3
Figure 12-3:	Resource use by cursor statement	12-4
Figure 12-4:	Read-only cursors and locking experiment input	12-11
Figure 12-5:	Update cursors and locking experiment input	12-12
Figure 13-1:	Worker process model.....	13-4
Figure 13-2:	Relative execution times for serial and parallel query execution.....	13-5
Figure 13-3:	A serial task scans data pages.....	13-7
Figure 13-4:	Worker processes scan an unpartitioned table	13-7
Figure 13-5:	Multiple worker processes access multiple partitions	13-8
Figure 13-6:	Hash-based, nonclustered index scan	13-9
Figure 13-7:	Join query using different parallel access methods on each table	13-10
Figure 13-8:	Steps for creating and loading a new partitioned table	13-18
Figure 14-1:	Parallel partition scan.....	14-5
Figure 14-2:	Parallel hash-based table scan	14-6
Figure 14-3:	Parallel clustered index partition scan	14-8
Figure 14-4:	Nonclustered index hash-based scan	14-10
Figure 14-5:	Worker process usage for joined tables	14-19
Figure 15-1:	Parallel sort strategy.....	15-4
Figure 15-2:	Area available for sort buffers	15-13
Figure 16-1:	How Adaptive Server uses memory.....	16-3
Figure 16-2:	The procedure cache.....	16-4
Figure 16-3:	Effect of increasing procedure cache size on the data cache	16-5
Figure 16-4:	Procedure cache size messages in the error log	16-5
Figure 16-5:	Formulas for sizing the procedure cache	16-6
Figure 16-6:	Effects of random selects on the data cache.....	16-9

Figure 16-7:	Effects of random data modifications on the data cache	16-10
Figure 16-8:	Formula for computing the cache hit ratio	16-11
Figure 16-9:	Caching strategies joining a large table and a small table	16-15
Figure 16-10:	Concepts in strict and relaxed LRU caches.....	16-21
Figure 16-11:	Victim and wash pointers moving in an LRU cache	16-23
Figure 16-12:	Fragmentation on a heap table	16-37
Figure 16-13:	Trade-offs in auditing and performance	16-42
Figure 17-1:	Physical and logical disks.....	17-4
Figure 17-2:	Spreading I/O across disks.....	17-5
Figure 17-3:	Joining tables on different physical devices.....	17-6
Figure 17-4:	Isolating database I/O from server-wide I/O.....	17-6
Figure 17-5:	Placing log and data on separate physical disks.....	17-8
Figure 17-6:	Disk I/O for the transaction log	17-8
Figure 17-7:	Mirroring data to separate physical disks.....	17-9
Figure 17-8:	Impact of mirroring on write performance.....	17-10
Figure 17-9:	Segment labeling a set of disks.....	17-11
Figure 17-10:	Separating a table and its nonclustered indexes	17-12
Figure 17-11:	Splitting a large table across devices with segments.....	17-13
Figure 17-12:	Placing the text chain on a separate segment	17-13
Figure 17-13:	Addressing page contention with partitions	17-17
Figure 17-14:	A table with 3 partitions on 3 devices	17-42
Figure 17-15:	Devices and partitions after create index.....	17-42
Figure 17-16:	Partitions almost completely fill the devices.....	17-43
Figure 17-17:	Extent stealing and unbalanced data distribution	17-43
Figure 18-1:	A kink in a page chain crossing allocation units.....	18-9
Figure 19-1:	tempdb default allocation	19-4
Figure 19-2:	tempdb spanning disks.....	19-10
Figure 19-3:	Optimizing and creating temporary tables.....	19-12
Figure 20-1:	Packet sizes and performance.....	20-5
Figure 20-2:	Reducing network traffic by filtering data at the server	20-7
Figure 20-3:	Effects of long transactions on other users.....	20-10
Figure 20-4:	Match network packet sizes to application mix.....	20-11
Figure 20-5:	Isolating heavy network users.....	20-12
Figure 20-6:	Configuring multiple network ports	20-13
Figure 21-1:	Process vs. subprocess architecture.....	21-4
Figure 21-2:	Processes queued in the run queue for a single CPU	21-5
Figure 21-3:	Multithreaded processing.....	21-5
Figure 21-4:	Tasks queue up for the Adaptive Server engine	21-7
Figure 21-5:	Task execution time schedule when other tasks are waiting	21-8
Figure 21-6:	Task execution time schedule when no other tasks are waiting.....	21-8
Figure 21-7:	A task fails to relinquish the engine within the scheduled time	21-9

Figure 21-8:	Processes queued in the OS run queue for multiple CPUs	21-11
Figure 21-9:	Tasks queue up for multiple Adaptive Server engines	21-12
Figure 21-10:	Scheduling model when execution classes are assigned	21-14
Figure 21-11:	Adaptive Server task management in the SMP environment.....	21-15
Figure 22-1:	Tasks queued in the three priority run queues.....	22-4
Figure 22-2:	An example of engine affinity.....	22-9
Figure 22-3:	Execution objects and their tasks.....	22-10
Figure 22-4:	How execution class and engine affinity affect execution.....	22-11
Figure 22-5:	Precedence rule	22-13
Figure 22-6:	Use of the precedence rule	22-14
Figure 22-7:	Conflict resolution	22-18
Figure 22-8:	Process for assigning execution precedence	22-24
Figure 24-1:	sp_sysmon execution algorithm.....	24-2
Figure 24-2:	Eliminating one bottleneck reveals another	24-8
Figure 24-3:	How Adaptive Server spends its available CPU time.....	24-13
Figure 24-4:	How transactions are counted	24-40
Figure 24-5:	Clustered table before inserts.....	24-53
Figure 24-6:	Insert causes a page split	24-53
Figure 24-7:	Another insert causes another page split.....	24-53
Figure 24-8:	Page splitting continues.....	24-54
Figure 24-9:	First insert with ascending inserts mode	24-54
Figure 24-10:	Additional ascending insert causes a page allocation.....	24-54
Figure 24-11:	Additional inserts fill the new page.....	24-55
Figure 24-12:	Cache management categories	24-67

List of Tables

Table 1:	Font and syntax conventions in this manual.....	xlix
Table 2:	Types of expressions used in syntax statements.....	li
Table 5-1:	Summary of locks for insert and create index statements.....	5-11
Table 5-2:	Summary of locks for select, update and delete statements.....	5-11
Table 5-3:	Lock compatibility.....	5-12
Table 5-4:	Access method and lock types.....	5-22
Table 6-1:	sp_spaceused output.....	6-5
Table 6-2:	dbcc commands that report space usage.....	6-7
Table 6-3:	Storage sizes for Adaptive Server datatypes.....	6-13
Table 7-1:	Tools for managing index performance.....	7-6
Table 7-2:	Additional tools for managing index performance.....	7-7
Table 7-3:	Advanced tools for query tuning.....	7-8
Table 7-4:	statistics io output for reads.....	7-11
Table 7-5:	Composite nonclustered index ordering and performance.....	7-37
Table 7-6:	Comparing index strategies for two queries.....	7-41
Table 7-7:	Default density percentages.....	7-45
Table 7-8:	Page pointers for unpartitioned tables in the sysindexes table.....	7-50
Table 8-1:	SARG equivalents.....	8-11
Table 8-2:	Default density percentages.....	8-19
Table 8-3:	Special access methods for aggregates.....	8-27
Table 8-4:	Effects of indexing on update mode.....	8-44
Table 9-1:	Basic showplan messages.....	9-2
Table 9-2:	showplan messages for various clauses.....	9-12
Table 9-3:	showplan messages describing access methods.....	9-23
Table 9-4:	showplan messages for parallel queries.....	9-39
Table 9-5:	showplan messages for subqueries.....	9-46
Table 9-6:	Internal subquery aggregates.....	9-53
Table 10-1:	Index name and prefetching.....	10-12
Table 10-2:	Optimizer hints for serial and parallel execution.....	10-16
Table 10-3:	Operators in dbcc traceon(302) output.....	10-21
Table 10-4:	Base cost output.....	10-23
Table 11-1:	Density approximations for unknown search arguments.....	11-3
Table 12-1:	Locks and memory use for isql and Client-Library client cursors.....	12-5
Table 12-2:	Sample execution times against a 5000-row table.....	12-10
Table 12-3:	Locks held on data and index pages by cursors.....	12-11
Table 12-4:	Effects of for update clause and shared on cursor locking.....	12-16
Table 13-1:	Configuration parameters for parallel execution.....	13-12
Table 13-2:	set options for parallel execution tuning.....	13-14

Table 13-3:	Scaling of engines and worker processes.....	13-20
Table 14-1:	Parallel access method summary	14-12
Table 14-2:	Determining applicable access methods	14-13
Table 15-1:	Number of producers and consumers used for create index.....	15-7
Table 15-2:	Basic sort resource messages.....	15-19
Table 16-1:	Commands used to configure caches	16-17
Table 16-2:	Differences between strict and relaxed LRU replacement policy	16-22
Table 16-3:	Effects of recovery interval on performance and recovery time.....	16-40
Table 17-1:	Assigning partitions to segments	17-18
Table 18-1:	Look-ahead set sizes.....	18-6
Table 20-1:	Network options	20-8
Table 22-1:	Fixed-attribute composition of predefined execution classes	22-4
Table 22-2:	System procedures for managing execution object precedence	22-7
Table 22-3:	Conflicting attribute values and Adaptive Server assigned values	22-17
Table 22-4:	Example analysis of an Adaptive Server environment.....	22-19
Table 22-5:	When assigning execution precedence is useful.....	22-21
Table 23-1:	Using the sorted_data option for creating a clustered index	23-5
Table 24-1:	<i>sp_sysmon</i> report sections	24-6
Table 24-2:	Action to take based on metadata cache usage statistics.....	24-59

About This Book

This book discusses performance and tuning issue for Sybase® Adaptive Server™ Enterprise.

Audience

This manual is intended for:

- Sybase System Administrators
- Database designers
- Application developers

How to Use This Book

This book contains the following chapters:

Chapter 1, “Introduction to Performance Analysis,” describes the major components to be analyzed when addressing performance.

Chapter 2, “Database Design and Denormalizing for Performance,” provides a brief description of relational databases and good database design.

Chapter 3, “Data Storage,” describes Adaptive Server page types, how data is stored on pages, and how queries on heap tables are executed.

Chapter 4, “How Indexes Work,” provides information on how indexes are used to resolve queries.

Chapter 5, “Locking in Adaptive Server,” describes locking in Adaptive Server and techniques for reducing lock contention.

Chapter 6, “Determining or Estimating the Sizes of Tables and Indexes,” describes different methods for determining the current size of database objects and for estimating their future size.

Chapter 7, “Indexing for Performance,” provides guidelines and examples for choosing indexes.

Chapter 8, “Understanding the Query Optimizer,” describes the operation of the query optimizer for serial queries.

Chapter 9, “Understanding Query Plans,” provides examples of `showplan` messages.

Chapter 10, “Advanced Optimizing Techniques,” describes advanced tools for tuning query performance.

Chapter 11, “Transact-SQL Performance Tips,” contains tips and workarounds for specific types of queries.

Chapter 12, “Cursors and Performance,” describes performance issues with cursors.

Chapter 13, “Introduction to Parallel Query Processing,” introduces the concepts and resources required for parallel query processing.

Chapter 14, “Parallel Query Optimization,” provides an in-depth look at the optimization of parallel queries.

Chapter 15, “Parallel Sorting,” describes the use of parallel sorting for queries and for creating indexes.

Chapter 16, “Memory Use and Performance,” describes how Adaptive Server uses memory for the procedure and data caches.

Chapter 17, “Controlling Physical Data Placement,” describes the uses of segments and partitions for controlling the physical placement of data on storage devices.

Chapter 18, “Tuning Asynchronous Prefetch,” describes how asynchronous prefetch improves performance for queries that perform large amounts of disk I/O.

Chapter 19, “tempdb Performance Issues,” stresses the importance of the temporary database, *tempdb*, and provides suggestions for improving its performance.

Chapter 20, “Networks and Performance,” describes network issues.

Chapter 21, “How Adaptive Server Uses Engines and CPUs,” describes how client processes are scheduled on engines in Adaptive Server.

Chapter 22, “Distributing Engine Resources Between Tasks,” describes how to assign execution precedence to specific applications.

Chapter 23, “Maintenance Activities and Performance,” describes the impact of maintenance activities on performance, and how some activities, such as re-creating indexes, can improve performance.

Chapter 24, “Monitoring Performance with *sp_sysmon*,” describes how to use a system procedure that monitors Adaptive Server performance.

Adaptive Server Enterprise Documents

The following documents comprise the Sybase Adaptive Server Enterprise documentation:

- 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 SyBooks™-on-the-Web.
- The Adaptive Server installation documentation for your platform – describes installation and upgrade procedures for all Adaptive Server and related Sybase products.
- The Adaptive Server configuration documentation for your platform – describes configuring a server, creating network connections, configuring for optional functionality, such as auditing, installing most optional system databases, and performing operating system administration tasks.
- *What's New in Adaptive Server Enterprise?* – describes the new features in Adaptive Server release 11.5, the system changes added to support those features, and the changes that may affect your existing applications.
- *Navigating the Documentation for Adaptive Server* – an electronic interface for using Adaptive Server. This online document provides links to the concepts and syntax in the documentation that are relevant to each task.
- *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 and user and system databases, and specifying character conversion, international language, and sort order settings.
- *Adaptive Server Reference Manual* – contains detailed information about all Transact-SQL commands, functions, procedures, and datatypes. This manual also contains a list of the Transact-SQL reserved words and definitions of system tables.

- *Performance and Tuning Guide* – explains how to tune Adaptive Server for maximum performance. This manual includes information about database design issues that affect performance, query optimization, how to tune Adaptive Server for very large databases, disk and cache issues, and the effects of locking and cursors on performance.
- The *Utility Programs* manual for your platform – documents the Adaptive Server utility programs, such as `isql` and `bcp`, which are executed at the operating system level.
- *Security Features User's Guide* – provides instructions and guidelines for using the security options provided in Adaptive Server from the perspective of the non-administrative user.
- *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 for Adaptive Server Enterprise and OmniConnect* – explains how to use the Adaptive Server Component Integration Services feature to connect remote Sybase and non-Sybase databases.
- *Adaptive Server Glossary* – defines technical terms used in the Adaptive Server documentation.
- *Master Index for Adaptive Server Publications* – combines the indexes of the *Adaptive Server Reference Manual*, *Component Integration Services User's Guide*, *Performance and Tuning Guide*, *Security Administration Guide*, *Security Features User's Guide*, *System Administration Guide*, and *Transact-SQL User's Guide*.

Other Sources of Information

Use the SyBooks™ and SyBooks-on-the-Web online resources to learn more about your product:

- SyBooks documentation is on the CD that comes with your software. The DynaText browser, also included on the CD, allows you to access technical information about your product in an easy-to-use format.

Refer to *Installing SyBooks* in your documentation package for instructions on installing and starting SyBooks.

- SyBooks-on-the-Web is an HTML version of SyBooks that you can access using a standard Web browser.

To use SyBooks-on-the-Web, go to <http://www.sybase.com>, and choose Documentation.

Conventions

The following 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 in this manual are as follows:

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</i> database
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 <i>row_aggregate</i> (<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}

Table 1: Font and syntax conventions in this manual (continued)

Element	Example
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.	<code>[extra_cheese, avocados, sour_cream]</code>
An ellipsis (...) means that you can repeat 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 "Changing the Default Character Set, Sort Order, or Language" in Chapter 13, "Configuring Character Sets, Sort Orders, and

Languages,” 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.

Basic Concepts

1

Introduction to Performance Analysis

This chapter introduces you to issues in performance tuning, and the layers at which the issues apply.

This chapter contains the following sections:

- What Is “Good Performance”? 1-1
- What Is Tuning? 1-2
- Know the System Limits 1-7
- Know Your Tuning Goals 1-7
- Steps in Performance Analysis 1-7

What Is “Good Performance”?

Performance is the measure of efficiency of 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. You can shorten response time 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:

- For a single transaction, for example, 5 UpdateTitle transactions per minute
- For 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.

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.

What Is Tuning?

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

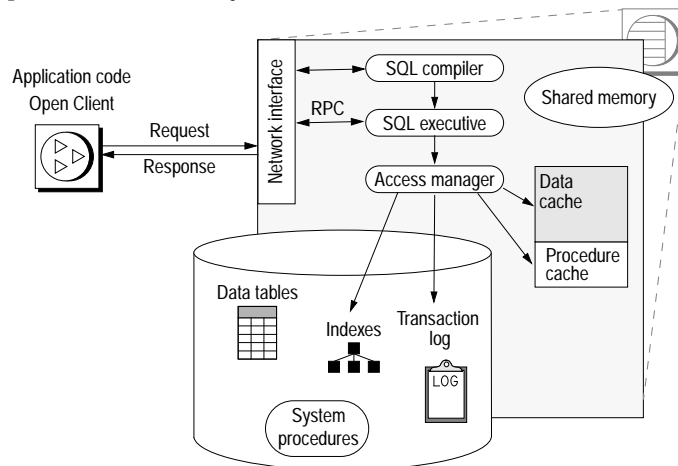


Figure 1-1: Adaptive Server system model

A major part of tuning is reducing contention for system resources. As the number of users increases, contention for resources such as

data and procedure caches, spinlocks on system resources, and the CPU(s) increases. The probability of lock contention on data pages also increases.

Tuning Levels

Adaptive Server and its environment and applications can be broken into components, or tuning layers, in order 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 tuning layers in Adaptive Server are:

- Applications layer – Most performance gains come from query tuning, based on good database design. Most of this guide is devoted to an explanation of Adaptive Server internals and query processing techniques and tools.
- Database layer – Applications share resources at the database layer, including disks, the transaction log, data cache.
- Server layer – At the server layer, there are many shared resources, including the data and procedure caches, locks, CPUs.
- Devices layer – The disk and controllers that store your data.
- Network layer – The network or networks that connect users to Adaptive Server.
- Hardware layer – The CPU(s) available.
- Operating system layer – Ideally, Adaptive Server is the only major application on a machine, and must only share CPU, memory, and other resources with the operating system, and other Sybase software such as Backup Server™ and Adaptive Server Monitor™.

Application Layer

Most of this guide describes tuning queries, since most of your efforts in maintaining high Adaptive Server performance will involve tuning the queries on your server.

Issues at the application layer include the following:

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

Options 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

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 include:

- Using transaction log thresholds to automate logs 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 parallelism
- Caching for high availability of critical tables and indexes

Adaptive Server Layer

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 option when the number of users and transaction rate reach high levels.

Options 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

Issues at the devices layer include the following:

- Will the master device, the devices that hold the user database, or the database logs be mirrored?
- How do you distribute system databases, user databases, and database logs across the devices?
- Are partitions needed for parallel query performance or high insert performance on heap tables?

Options include:

- Using more medium-sized devices and more 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

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 include:

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

Hardware Layer

Issues at the hardware layer include:

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

Options 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

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

Know the 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 purchasing more memory and faster components. Examples are 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.

Know Your 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.

Steps in Performance Analysis

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 Chapter 24, “Monitoring Performance with `sp_sysmon`,” 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 the following:
 - 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 an iterative 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 was correct, implement the solution in your development environment.

Using *sp_sysmon* to Monitor Performance

Use the system procedure *sp_sysmon* while you are tuning to monitor the effects of adjustments you make.

Performance tuning is usually an iterative process. While specific tuning might enhance performance in one area, it can simultaneously diminish performance in another area. Check the entire *sp_sysmon* output and make adjustments as necessary to achieve your tuning goals.

For more information about using *sp_sysmon* see Chapter 24, "Monitoring Performance with *sp_sysmon*."

Adaptive Server Monitor, a separate Sybase product, can pinpoint problems at the object level.

2

Database Design and Denormalizing for Performance

The chapter provides a brief description of database design and denormalization, and how these relate to performance. Performance and tuning are built on top of good database design. They aren't panaceas. If you start with a bad database design, the information in the other chapters of this book may help you speed up your queries a little, but good overall performance starts with good design.

This chapter does not attempt to discuss all of the material presented in database design courses. It cannot teach you nearly as much as the many excellent books available on relational database design. This chapter presents some of the major design concepts and a few additional tips to help you move from a logical database design to a physical design on Adaptive Server.

This chapter contains the following sections:

- Database Design 2-1
- Normalization 2-2
- Denormalizing for Performance 2-6

Database Design

Database design is the process of moving from real-world business models and requirements to a database model that meets these requirements. 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).

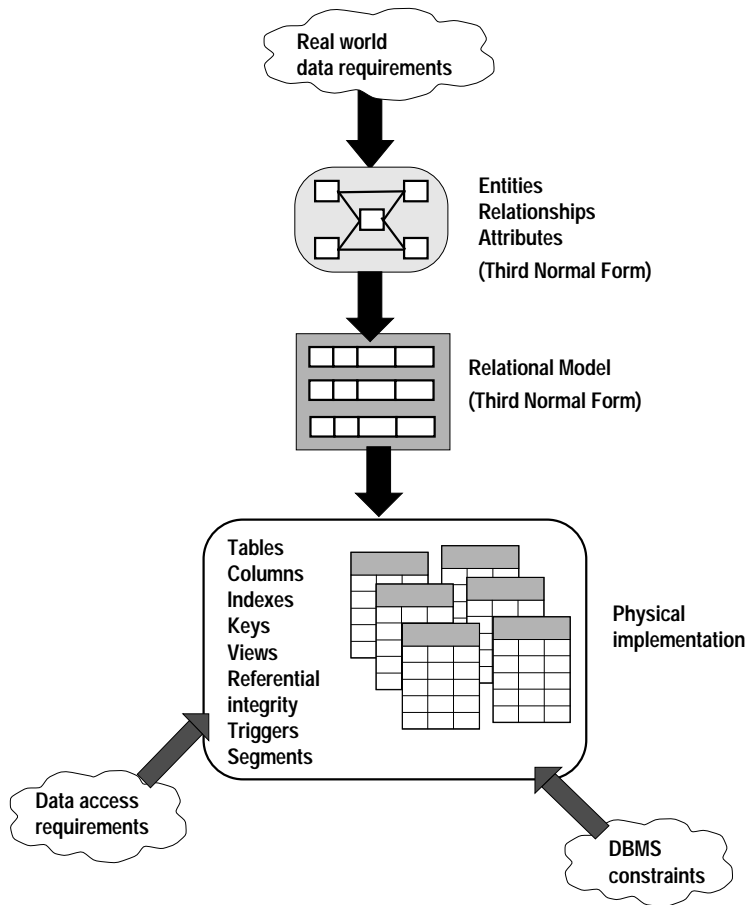


Figure 2-1: Database design

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

Normalization

When a table is normalized, the non-key columns depend on the key, the whole key, and nothing but the key.

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 teams of developers.

However, a fully normalized design may not always yield the best performance. It is recommended that you design for Third Normal Form, and then, as performance issues arise, denormalize to solve them.

Levels of Normalization

Each level of normalization relies on the previous level, as shown in Figure 2-2. For example, to conform to 2NF, entities must be in 1NF.

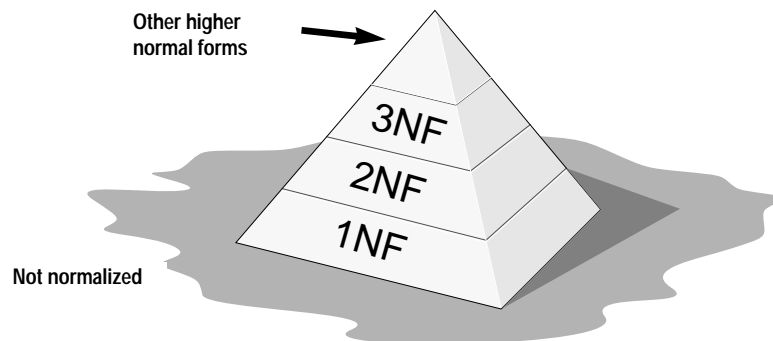


Figure 2-2: Levels of normalization

When determining if a database is in a normal form, start with the assumption that the relation (or table) is not normalized. Then apply the rigor of each normal form level to it.

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 are faster, since tables are narrower, and more rows fit on a data page.
- You usually wind up with more tables. You can have more clustered indexes (you get only 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 wind up with fewer indexes per table, so data modification commands are faster.
- You wind up with 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.
- Every row and column position can have only one value.

The table in Figure 2-3 violates first normal form, since the *dept_no* column contains a repeating group:

Employee (emp_num, emp_lname, dept_no)

Employee		
emp_num	emp_lname	dept_no
10052	Jones	A10 C66
10101	Sims	D60

Repeating group

Figure 2-3: A table that violates first normal form

Normalization creates two tables and moves *dept_no* to the second table:

Employee (emp_num, emp_lname)

Emp_dept (emp_num, dept_no)

Employee		Emp_dept	
emp_num	emp_lname	emp_num	dept_no
10052	Jones	10052	A10
10101	Sims	10052	C66
		10101	D60

Figure 2-4: Correcting first normal form violations by creating two tables

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 the table in Figure 2-5, 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.

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

Primary key

Depends on part of primary key

Figure 2-5: A table that violates second normal form

To normalize this table, move *dept_name* to a second table, as shown in Figure 2-6.

Emp_dept (emp_num, dept_no)

Emp_dept	
emp_num	dept_no
10052	A10
10074	A10
10074	D60

Primary key

Dept (dept_no, dept_name)

Dept	
dept_no	dept_name
A10	accounting
D60	development

Primary key

Figure 2-6: Correcting second normal form violations by creating two tables

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 2-7 violates Third Normal Form because the *mgr_lname* field depends on the *mgr_emp_num* field, which is not a key field.

Dept (dept_no, dept_name, mgr_emp_num, mgr_lname)

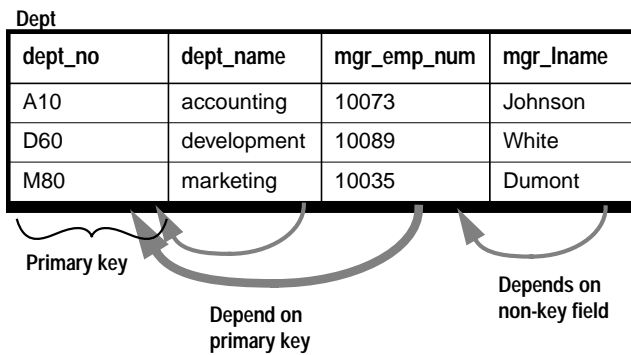


Figure 2-7: A table that violates Third Normal Form

The solution is to split the *Dept* table into two tables, as shown in Figure 2-8. In this case, the *Employees* table, shown in Figure 2-4 on page 2-5 already stores this information, so removing the *mgr_lname* field from *Dept* brings the table into Third Normal Form.

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 key

Employee (emp_num, emp_lname)

Employee

emp_num	emp_lname
10073	Johnson
10089	White
10035	Dumont

Primary key

Figure 2-8: Correcting Third Normal Form violations by creating two tables

Denormalizing for Performance

Once you have created your database in normalized form, you can perform benchmarks and decide to back away from normalization to improve performance for specific queries or applications.

The process of denormalizing:

- Can be done with tables or columns
- Assumes prior normalization
- Requires a thorough knowledge of how the data is being used

Good reasons for denormalizing are:

- 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 of Denormalization

Denormalization should be based on thorough knowledge of the application, and it should be performed 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. The same values can be obtained using this query:

```
select title_id, sum(qty)
  from salesdetail
  group by title_id
```

To obtain the summary values and the document title requires a join with the *titles* table, use this query:

```
select title, sum(qty)
  from titles t, salesdetail sd
  where t.title_id = sd.title_id
  group by title
```

It makes sense to denormalize this table if the query is run frequently. 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.

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.

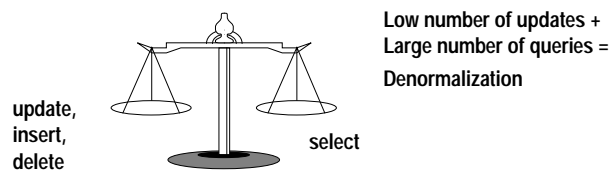


Figure 2-9: Balancing denormalization issues

Whatever form of denormalization you choose, it has the potential for data integrity problems that must be carefully documented and addressed in application design.

Disadvantages of Denormalization

Denormalization has these disadvantages:

- It usually speeds retrieval but can slow data modification.
- It is always application-specific and needs to be re-evaluated 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 of Denormalization

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.

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?

Denormalization Techniques

The most prevalent denormalization techniques are:

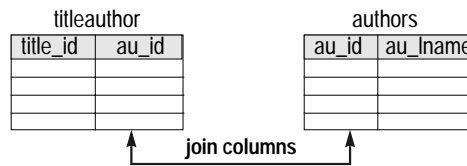
- Adding redundant columns
- Adding derived columns
- Collapsing tables

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 frequent joins are performed on the *titleauthor* and *authors* tables in order to retrieve the author's last name, you can add the *au_lname* column to *titleauthor*.

```
select ta.title_id, a.au_id, a.au_lname
from titleauthor ta, authors a
where ta.au_id = a.au_id
```



```
select title_id, au_id, au_lname
from titleauthor
```

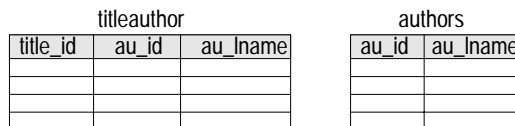


Figure 2-10: Denormalizing by adding redundant columns

Adding redundant columns eliminates joins for many queries. The problems with this solution are that it:

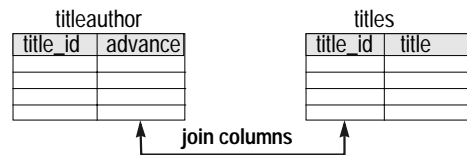
- Requires maintenance of new columns. All changes must be made to two tables, and possibly to many rows in one of the tables.
- Requires more disk space, since *au_lname* is duplicated.

Adding Derived Columns

Adding derived columns can help eliminate 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 2-11 shows both benefits. Frequent joins are needed between the *titleauthor* and *titles* tables to provide the total advance for a particular book title.

```
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
```

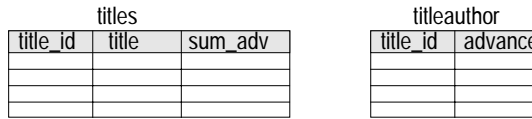


Figure 2-11: Denormalizing by adding derived columns

You can create and maintain a derived data column in the *titles* table, eliminating both the join and the aggregate at run time. 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.

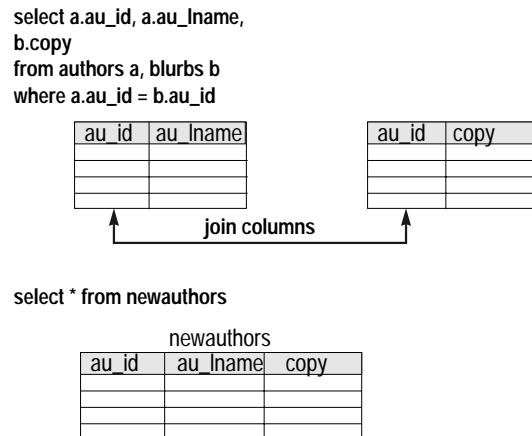


Figure 2-12: Denormalizing by collapsing tables

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.

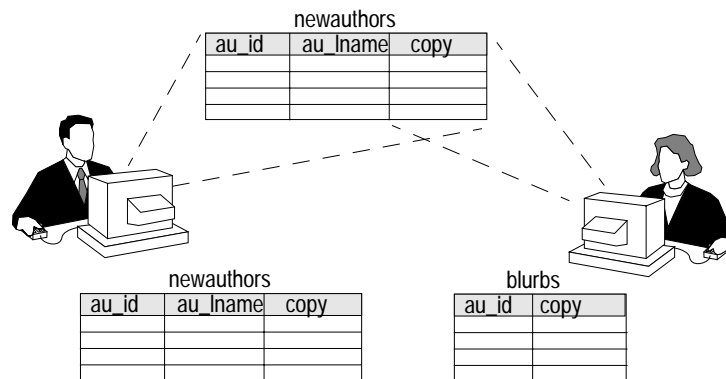


Figure 2-13: Denormalizing by duplicating tables

The kind of split shown in Figure 2-13 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.

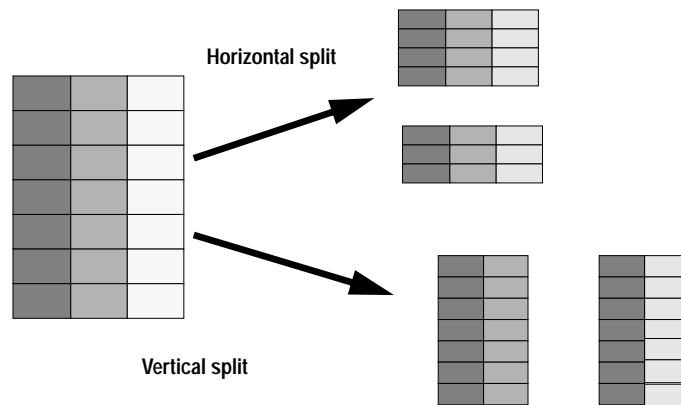


Figure 2-14: Horizontal and vertical partitioning of tables

Splitting tables—either horizontally or vertically—adds complexity to your applications. There usually needs to be a very good performance reason.

Horizontal Splitting

Use horizontal splitting in the following circumstances:

- 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 vs. 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 (there are other ways to accomplish this goal, too).

Generally, horizontal splitting adds a high degree of complication to applications. It usually requires different table names in queries, depending on values in the tables. This complexity alone usually far outweighs the advantages of table splitting in most database applications. 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 2-15 shows how the *authors* table might be split to separate active and inactive authors:

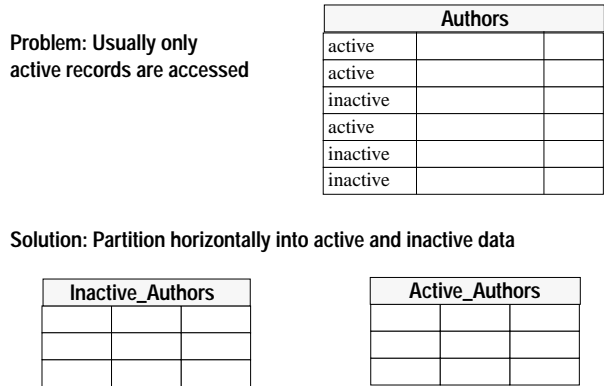


Figure 2-15: Horizontal partitioning of active and inactive data

Vertical Splitting

Use vertical splitting in the following circumstances:

- 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 not accessed frequently, 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 fewer pages can be accessed for many queries.

Figure 2-16 shows how the *authors* table can be partitioned.

Problem:

Frequently access lname and fname,
infrequently access phone and city

Solution: Partition data vertically

Authors				
au_id	lname	fname	phone	city

Authors_Frequent		
au_id	lname	fname

Authors_Infrequent		
au_id	phone	city

Figure 2-16: Vertically partitioning a table

Managing Denormalized Data

Whatever denormalization techniques you use, you need to develop management techniques that ensure data integrity. Choices include:

- 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 be sure 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 to Manage Denormalized Data

In Figure 2-17, 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.

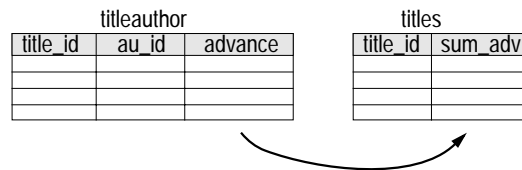


Figure 2-17: Using triggers to maintain normalized data

Using Application Logic to Manage Denormalized Data

If your application has to ensure data integrity, it will have to ensure that the inserts, deletes, or updates to both tables occur in a single transaction (see Figure 2-18).

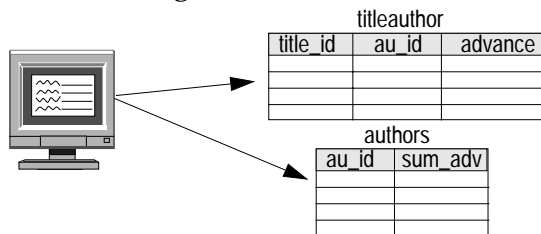


Figure 2-18: Maintaining denormalized data via application logic

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.

You can run short, frequent batches or long, in frequent batches (see Figure 2-19).

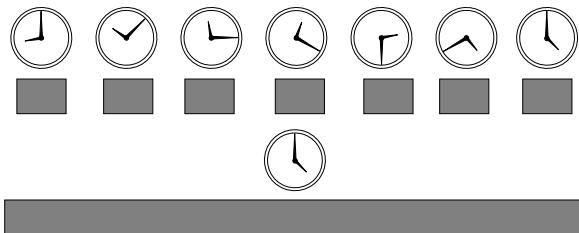


Figure 2-19: Using batch reconciliation to maintain data

3

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.

This chapter contains the following sections:

- Major Performance Gains Through Query Optimization 3-1
- Query Processing and Page Reads 3-2
- Adaptive Server Data Pages 3-3
- Additional Page Types 3-7
- The sysindexes Table and Data Access 3-10
- Heaps of Data: Tables Without Clustered Indexes 3-10
- How Adaptive Server Performs I/O for Heap Operations 3-14
- Caches and Object Bindings 3-14
- Heaps: Pros and Cons 3-19
- Maintaining Heaps 3-19
- The Transaction Log: A Special Heap Table 3-20

Major Performance Gains Through Query Optimization

Most of a query's execution time is spent reading data pages from disk. Therefore, most of your performance improvement—more than 80 percent, according to many performance and tuning experts—comes from reducing the number of disk reads needed for each query.

A table scan is the worst case: if 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 you need. The individual query has very poor response time, because disk reads take time.

Queries that incur 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.

Clearly, table scans maximize the number of disk reads (I/Os) for a given query. How can you predict disk reads? When you have become thoroughly familiar with the tools, the indexes on your tables, and the size and structure of the objects in your applications, you should be able to estimate the number of I/O operations a given join or select operation will perform. If you know the indexed columns on your tables and the table and index sizes, you can often look at a query and predict its behavior. For different queries on the same table, you might make these kinds of statements (ordered from best to worst in performance):

- “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 nonclustered index. This query will probably perform a scan on the leaf level of the index, about 600 pages. If I add an unindexed column to the select list, it has to scan the table, and that 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.”

Later 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.

Query Processing and Page Reads

Each time you submit a Transact-SQL query, the Adaptive Server optimizer determines the optimal access path to the needed data. In most database applications, you have many tables in the database, and each table has one or more indexes. The 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. 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 only a half-dozen page reads in all
- Index covering – using only a nonclustered 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 right 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.

Adaptive Server Data Pages

The basic unit of storage for Adaptive Server is a **page**. On most systems, a page is 2K, 2048 bytes. A page contains 32 bytes of header information. The rest of the page is available to store data rows and row pointers (the row offset table).

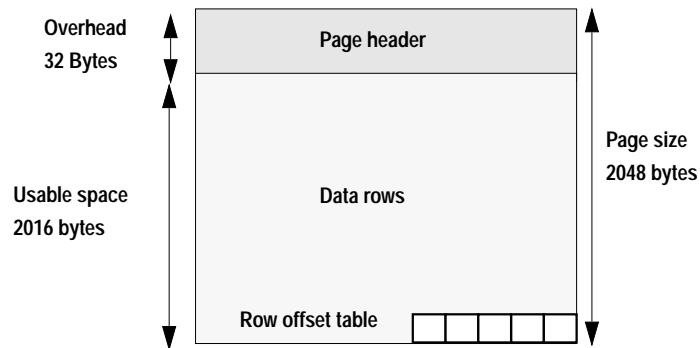


Figure 3-1: An Adaptive Server data page

Page headers use 32 bytes, leaving 2016 bytes for data storage on each page.¹ Information in the page header includes pointers to the next page and the previous page used by the object, and the object ID of the table or index using that page.

Each row is stored contiguously on the page. The information stored for each row consists of the actual column data plus information such as the row number (1 byte) and the number of variable-length and null columns in the row (1 byte).

1. The maximum number of bytes for a data row is 1960 (plus two bytes of overhead) due to overhead for logging: the row, plus the overhead about the transaction, must fit on a single page in the transaction log.

Rows cannot cross page boundaries, except for *text* and *image* columns. Each data row has at least 4 bytes of overhead; rows that contain variable-length data have additional overhead. See Chapter 6, “Determining or Estimating the 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. Each pointer requires 2 bytes.

Row Density on Data Pages

The usable space on a page, divided by the row size, tells us how many rows can be stored on a page. This figure gives us the row density. The size of rows can affect your performance dramatically: the smaller the data rows, the more rows you can store per page. When rows are small, Adaptive Server needs to read fewer pages to answer your select queries, so your performance will be better for queries that perform frequent table scans.

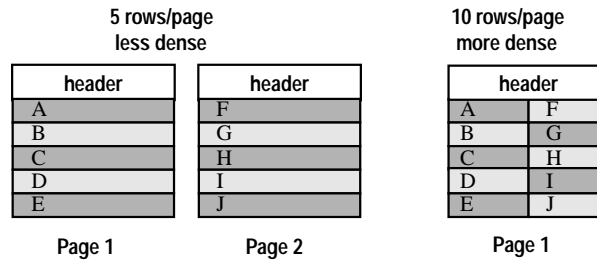


Figure 3-2: Row density

Row density can sometimes have a negative impact on throughput when data is being modified. If one user changes data in a row, the page is locked until the transaction commits. Other users cannot access the changed row or any other data on the page until the lock is released. Adaptive Server allows you to specify the maximum number of rows on a page for tables where such lock contention is a problem. See “Reducing Lock Contention with `max_rows_per_page`” on page 5-33 for more information.

If your table contains variable-length fields, the row size depends on the actual length of the data, so row density can vary from page to page.

Extents

Adaptive Server pages are always allocated to a database object, such as a table or an index, in blocks of 8 pages at a time. This block of 8 pages is called an **extent**. The smallest amount of space that a table or index can occupy is 1 extent, or 8 data pages. Extents are deallocated only when all the pages in an extent are empty.

See Figure 3-5 on page 3-9 for an illustration of extents and object storage.

In most cases, the use of extents in Adaptive Server is transparent to the user. One place where information about extents is visible is in the output from `dbcc` commands that check allocation. These commands report information about objects and the extents used by the objects.

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.

Linked Data Pages

Each table and each level of each index forms a doubly-linked list of pages. Each page in the object 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 a table, it reads the pages in order, following these page pointers.

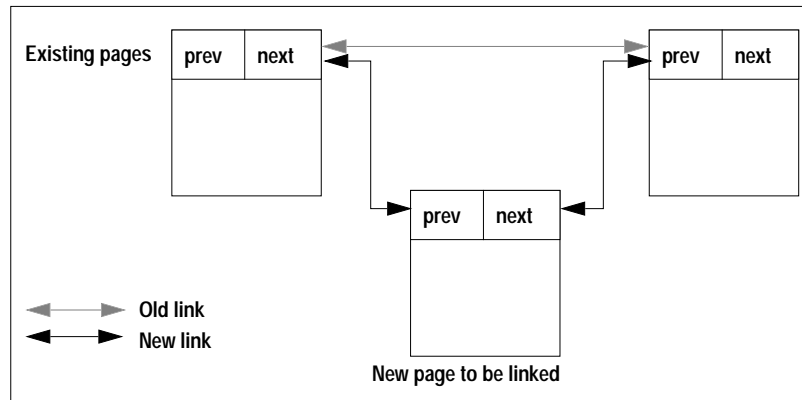


Figure 3-3: Page linkage

Adaptive Server tries to keep the page allocations close together for objects, as follows:

- 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.

For information on how page allocation is performed for partitioned tables, see "Commands for Partitioning Tables" on page 17-21.

Text and Image Pages

Text and image columns for a table are stored as a separate page chain, consisting of a set of text or image pages. If a table has multiple text or image columns, it still has only one of these separate data structures. Each table with a text or image column has one of these structures. The table itself stores a 16-byte pointer to the first page of the text value for the row. Additional pages for the value are linked by next and previous pointers, just like the data pages. Each *text* or *image* value is stored in a 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.

Figure 3-4 shows a table with text values. Each of the three rows stores a pointer to the starting location of its text value in the text/image structure.

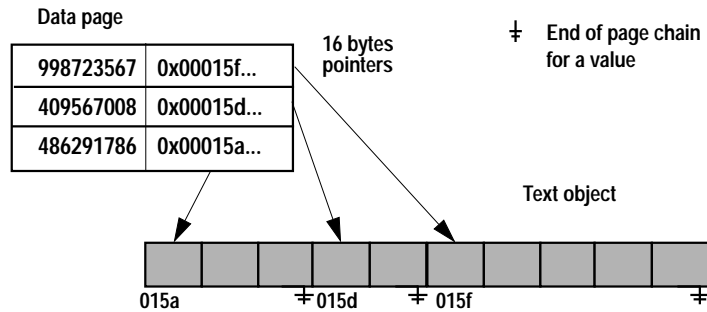


Figure 3-4: Text and image data storage

Reading or writing a text or image 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 text or image page stores up to 1800 bytes. Every non-null value uses at least one full data page.

Text objects are listed separately in *sysindexes*. The index ID column, *indid*, is always 255, and the *name* is the table name, prefixed with the letter “t”.

Additional Page Types

In addition to the page types discussed above for table storage, Adaptive Server uses index pages and several other types. Indexes are discussed in detail in the next chapter, and distribution pages are discussed in “How the Optimizer Uses the Statistics” on page 7-42.

This section describes other types of pages used by Adaptive Server to track space allocation and the location of database objects. These page types are mainly used to expedite the process of allocating and deallocating space for objects. They provide a way for Adaptive Server to allocate additional space for an object that is near the space

already used by the object. This strategy also 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, which contain allocation bitmaps for an entire database.
- Allocation pages, which track space usage and objects within groups of 256 database pages, or 1/2MB.
- Object Allocation Map (OAM) pages, which 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, which are used to manage space allocation for partitioned tables. Each partition has one control page. For more information on control pages, see “Effects of Partitioning on System Tables” on page 17-18.

Global Allocation Map (GAM) Pages

Each database has a GAM page. 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. The allocation page tracks space usage by objects on all of the extents in the unit by recording the object IDs and tracking what pages are in use and what pages are free in each extent. For each extent, it also stores the page ID for the OAM page of the object stored on that extent.

`dbcc` allocation-checking commands report space usage by allocation unit. Page 0 and all pages that are multiples of 256 are allocation pages.

Object Allocation Map (OAM) Pages

Each table, index, and text chain has one or more 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. These OAM pages store pointers to each allocation unit that contains 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, keeping the new space close to pages already used by the object.

Each OAM page holds allocation mappings (OAM entries) for 250 allocation units. A single OAM page stores information for 2000–63750 data or index pages.

Why the Range?

Each entry in the OAM page stores the page ID of the allocation page and the number of free and used pages for the object within that allocation page. If a table is spread out across the storage space for a database so that each allocation unit stores only one extent (8 pages) for the table, the 250 rows on the OAM page can only point to $250 * 8 = 2000$ database pages. If the table is very compactly stored, so that it uses all 255 pages available in each of its allocation units, 1 OAM page stores allocation information for $250 * 255 = 63,750$ pages.

Relationships Between Objects, OAM Pages, and Allocation Pages

Figure 3-5 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 object IDs 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. Of course, these allocation pages also

point to other objects stored in the allocation unit, but these pointers are not shown here.

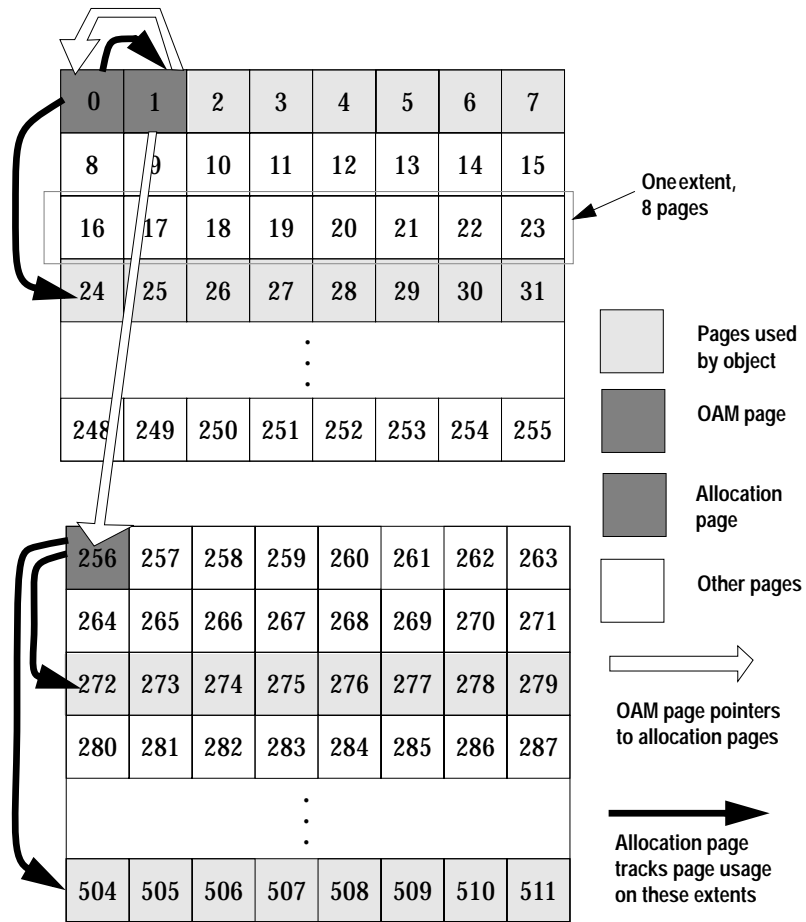


Figure 3-5: OAM page and allocation page pointers

The *sysindexes* Table and Data Access

The *sysindexes* table stores information about indexed and unindexed tables. *sysindexes* has one row:

- For each table
 - If the table has a clustered index, the *indid* column is 1; if it has no clustered index, the *indid* column is 0.
- For each nonclustered index
 - The index IDs are between 2 and 250.
- Each table with one or more *text* or *image* columns
 - The index ID is always 255.

Each row in *sysindexes* has three columns that help optimize access to objects:

Column	Description
<i>root</i>	Stores the page number of the root page of the index. If the row describes a table without a clustered index, this row stores a pointer to the last page of the heap. For partitioned heaps, this value is ignored; the <i>syspartitions</i> table stores the pointers to the last page for each partition. See Chapter 17, "Effects of Partitioning on System Tables," for more information.
<i>first</i>	Stores the page number of the first page of the data page chain, for a table, or the first page of the leaf level, for a nonclustered index.
<i>distribution</i>	Stores the page number of the page where statistics about index keys are stored.

When Adaptive Server needs to access a table, it uses these columns in *sysindexes* as a starting point.

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, and new data is always inserted on the last page of the table.

This section describes how select, insert, delete, and update operations perform on heaps when there is no nonclustered index to aid in retrieving data.

Select Operations on Heaps

When you issue a select operation 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. Adaptive Server must examine every row on every page in the table. Adaptive Server reads the *first* column in *sysindexes* for the table, reads the first page into cache, and then follows the next page pointers until it finds the last page of the table.

```
select * from employee
where emp_id = 12854
```

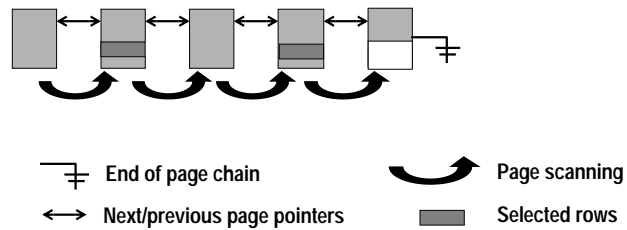


Figure 3-6: Selecting from a heap

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 you name in your *where* clause, but the optimizer determines that it would be more costly to use the index than to scan the table. Later chapters 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 also used when you issue a *select* without a *where* clause so that 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 4-17.

Inserting Data into a Heap

When you insert data into a heap, the data row is always added to the last page of the table. If the last page is full, a new page is allocated in the current extent. 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.

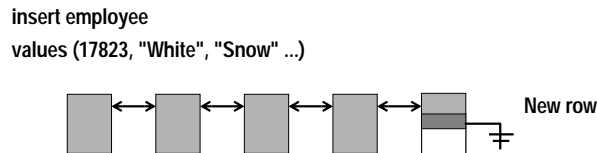


Figure 3-7: Inserting a row into a heap table

Adaptive Server allows you to specify the maximum number of rows on a page. If you use this option, a heap page is “full” when you have inserted that many rows on the page, and a new page is allocated. See “Reducing Lock Contention with `max_rows_per_page`” on page 5-33 for more information.

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. In addition, a special bit in the page header enables Adaptive Server to track whether row inserts are taking place continuously at the end of the page and disables the normal page-split mechanism. See “Page Splitting on Full Data Pages” on page 4-7 for more information.

One of the severe performance limits on heap tables is that the page must be locked when the row is added. If many users are trying to add rows to a heap table at the same time, they will block each other’s access to the page. These characteristics of heaps are true for:

- Single row inserts using `insert`
- Multiple row inserts using `select into` or `insert...select` (an insert statement that selects rows from another table or from the same table)
- Bulk copy into the table

In many cases, creating a clustered index for the table solves these performance problems for heaps and provides real performance gains for user queries. Another workaround for the last-page problem in heaps is to use partitions to create many “last pages” for the heap table. See “Improving Insert Performance with Partitions” on page 17-15 for more information.

Deleting Data from a Heap

When you delete rows from a heap, 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.

When a data row is deleted from the page, the rows that follow it on the page move up so that the data on the page remains contiguous.

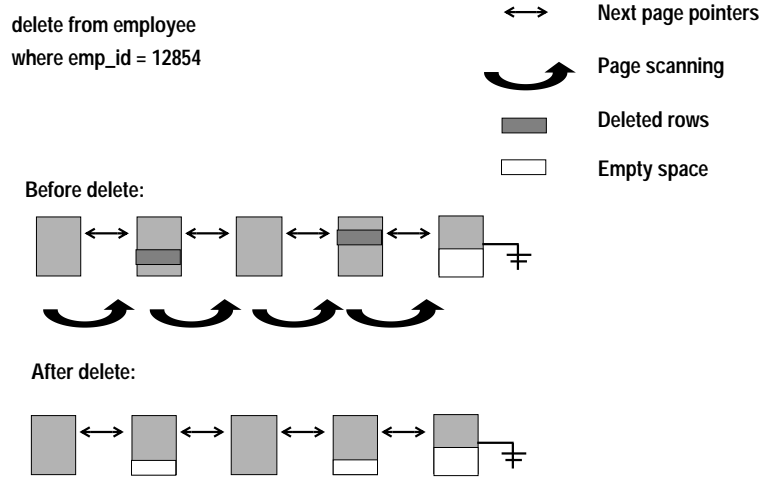


Figure 3-8: Deleting rows from a heap table

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 whole 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.

Update Operations on Heaps

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.

Updates on heaps 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 contention on the last page of the heap, just as inserts do.

For more information on how updates are performed, see “Update Operations” on page 8-32.

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 and other objects to the caches
- Configure data caches to perform I/O in page-sized multiples, up to eight data pages (one extent)

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

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

Sequential Prefetch, or Large I/O

Adaptive Server’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 can contain pools of 2K, 4K, 8K, and 16K buffers, allowing Adaptive Server to read up to an entire extent (eight data pages) in a single I/O operation. 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, all pages are written to disk

as a unit. 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. If your queries are performing table scans, as described in this chapter, you will often see great performance gains using large I/O.

For more information on configuring memory caches for large I/O, See Chapter 16, “Memory Use and Performance.”

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 cache. The default cache can also have buffer pools configured for large I/O. If your applications include some heap tables, they will probably perform better when bound to a cache that allows large I/O or when the default cache is configured for large 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 pages in the cache that have been changed pass a point, called the **wash marker**, on the MRU/LRU chain, Adaptive Server initiates an asynchronous write on the page. This ensures 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 and MRU, or **fetch-and-discard** replacement strategy.

LRU Replacement Strategy

LRU replacement strategy reads the data pages sequentially into the cache, replacing a “least recently used” buffer. Using strict LRU policy, the buffer is placed on the MRU end of the data buffer chain.

It moves down the cache toward the LRU end as more pages are read into the cache.

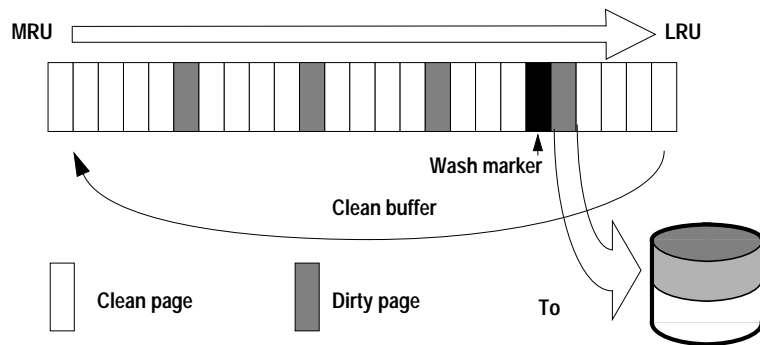


Figure 3-9: Strict LRU strategy takes a clean page from the LRU end of the cache

Relaxed LRU replacement policy does not maintain the pages in MRU/LRU order. It maintains a pointer to a victim buffer that is a set number of pages away from the wash marker. The new page is read into the victim buffer.

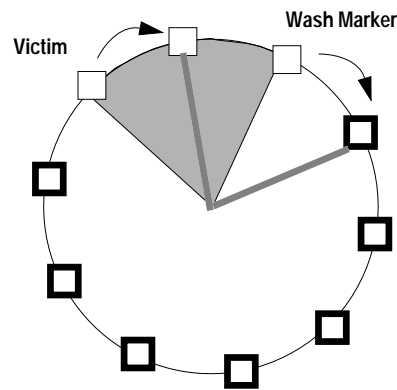


Figure 3-10: Relaxed LRU cache policy

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
- Many index pages
- Any query where LRU strategy is specified

MRU Replacement Strategy

MRU (fetch-and-discard) replacement strategy is often used for table scanning on heaps. This strategy reads a page into the cache just before the wash marker.

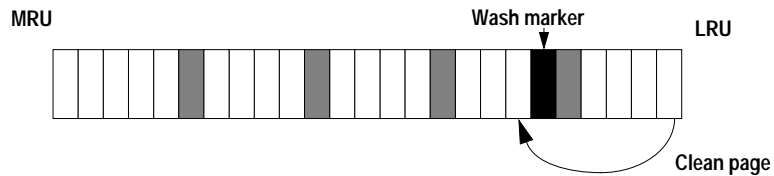


Figure 3-11: MRU strategy places pages just before the wash marker

Fetch-and-discard is most often used for queries where a page is needed only once by the query. This includes:

- Most table scans of large heaps in queries that do not use joins
- One or more tables in certain joins

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.

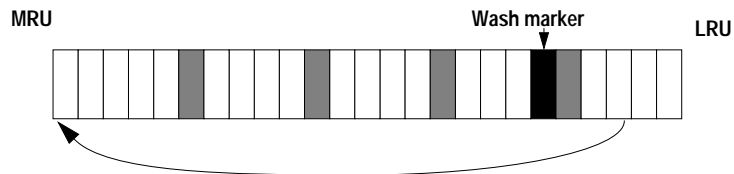


Figure 3-12: Finding a needed page in cache

Adaptive Server usually uses the fetch-and-discard strategy when it scans a large heap table and the table is not the inner table of a join. Each page for the table is needed only once. If the LRU strategy were used, the pages would move to the top of the MRU chain and force other pages out of cache.

Select Operations and Caching

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

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

For heaps, select operations performing extent-sized 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 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 heaps is effective only when the page chains are not fragmented. See “Maintaining Heaps” on page 3-19 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. Their changes are logged in the transaction log, but the changed pages are not written to disk immediately.

Caching and Inserts on Heaps

Inserts on heaps take place on the last page of the heap table. 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 3-13. 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.

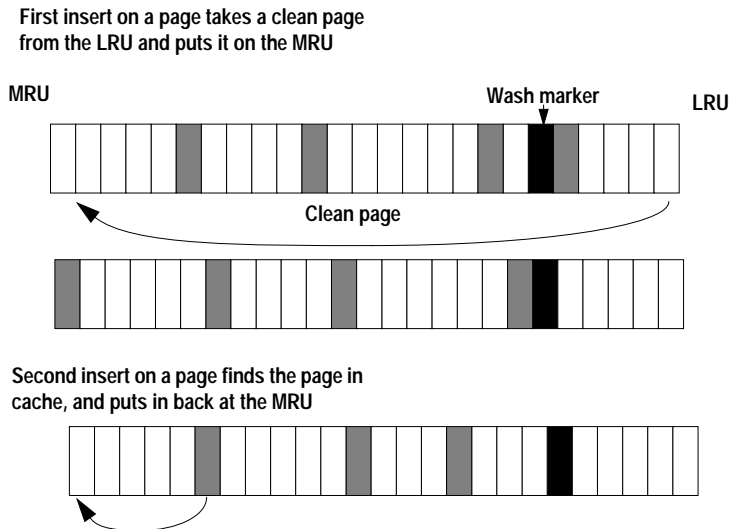


Figure 3-13: Inserts to a heap page in the data cache

The changed data page remains in cache until it moves past the wash marker or until a checkpoint or the housekeeper task writes it to disk. “The Data Cache” on page 16-6 explains more about these processes.

Caching and 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 whole buffer (a single page, or more, depending on the I/O size) is placed on the MRU end of the chain, and the row is changed. If the page is not in cache, it is read from the disk into a clean buffer from the LRU end of the cache and examined to determine whether the rows on the page match query clauses. Its placement on the MRU/LRU chain depends on whether any 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 be 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.

Heaps: Pros and Cons

Sequential disk access is efficient, however, the entire table must always be scanned to find any value.

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 (unless the heap table is partitioned).

Heaps work well for small tables and for tables where changes are infrequent, but they do not work well for large tables.

Guidelines for Using Heaps

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
- Have nonunique rows and the above characteristics
- Do not have large numbers of inserts and updates

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. Deletes and updates:

- Can result in many partially filled pages
- Can lead to inefficient large I/O, since page chains will not be contiguous on the extents

Methods for Maintaining Heaps

There are two methods to reclaim space in heaps after deletes and updates have created empty space on pages or have caused fragmentation:

- Create and then drop a clustered index
- Use `bcp` (the bulk copy utility) and `truncate table`

Reclaiming Space by Creating a Clustered Index

You can create and drop a clustered index on a heap table in order 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 percent of the table size. Since the leaf level of the clustered index consists of the actual data rows of the table, the process of creating the index makes a complete copy of the table before it deallocates the old pages. The additional 20 percent provides room for the root and intermediate index levels. If you use long keys for the index, it will take more space.

Reclaiming Space Using *bcp*

The steps to reclaim space with `bcp` are:

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 17-30 for procedures for working with partitioned tables. For more information on `bcp`, see the *Utility Programs* manual for your platform.

The 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.

Later 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 Chapter 15, “Creating and Managing User Databases,” in 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.

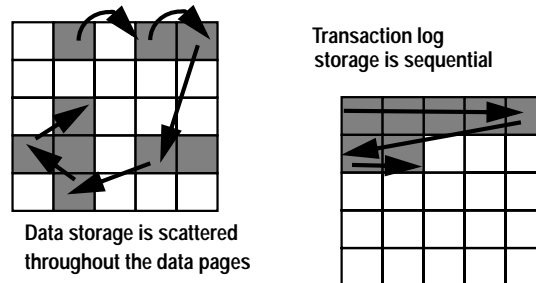


Figure 3-14: Data vs. log I/O

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.

4

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. This chapter contains the following sections:

- From Heaps of Pages to Fast Performance 4-1
- Types of Indexes 4-2
- Clustered Indexes 4-4
- Nonclustered Indexes 4-12
- Index Covering 4-17
- Indexes and Caching 4-19

From Heaps of Pages to Fast Performance

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.

Indexing requires trade-offs. 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

What Are Indexes?

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 (see Figure 4-1).

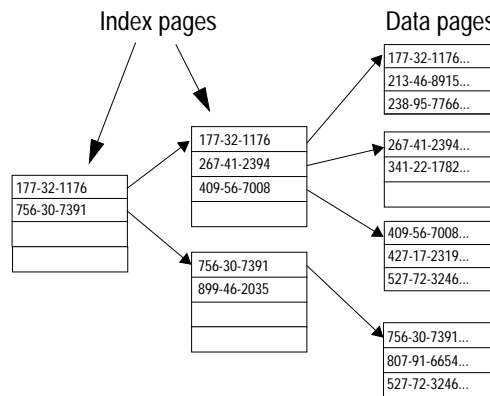


Figure 4-1: A simplified index schematic

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
- Nonclustered indexes, where the storage order of data in the table is not related to index keys

You can only create 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 3, “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 (for clustered indexes) or to the data rows (for the leaf level of nonclustered indexes). 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. A data row might have 200 bytes (including row overhead), so there would be 10 rows per page. An index on a 15-byte field would have 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 the table is very small, so that the entire index fits on a single page, there are no intermediate levels, and the root page stores pointers to 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, the leaf level is the data. No other level of the index contains one index row for each data row.
- For nonclustered indexes, the leaf level contains the index key values, a pointer to the page where the rows are 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 table may not have an intermediate level at all. In this case, the root pages point directly to the leaf level. Each level (except the root level) of the index is a page chain: The page headers contain next page and previous page pointers to other pages at the same index level (see Figure 4-2).

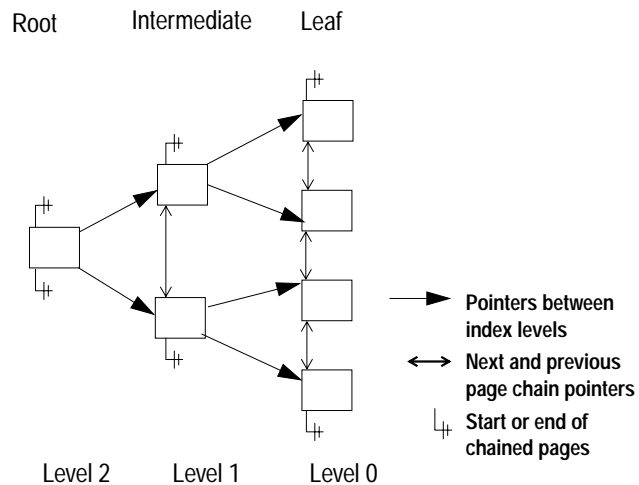


Figure 4-2: Index levels and page chains

For nonclustered indexes, the leaf level is always level 0. In clustered indexes, the level just above the data level is level 0. Each higher level is numbered sequentially, with the root page having the highest value.

B-trees are self-maintaining structures that obtain additional space as needed, without having to reorganize pages.

Clustered Indexes

In clustered indexes, leaf-level pages are also the data pages. The data rows are physically ordered by the index key. Physical ordering means that:

- All entries on a page are in index key order
- By following the “next page” pointers at the data level, you read the entire table in index key order

On the root and intermediate pages, each entry points to a page on the next level (see Figure 4-3).

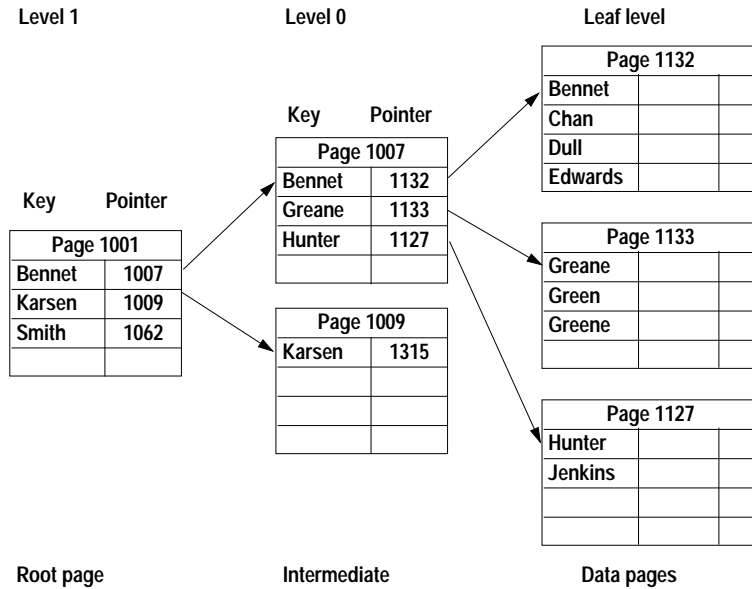


Figure 4-3: Clustered index on last name

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. In Figure 4-4, there is a clustered index on the “last name” column.

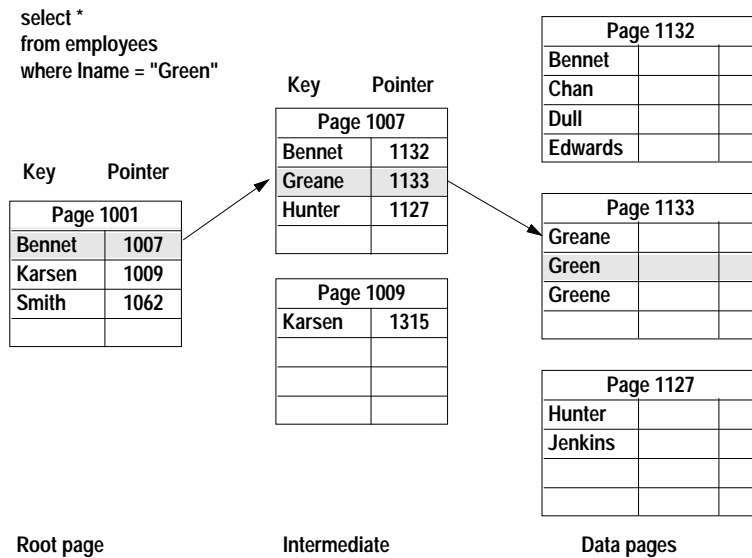


Figure 4-4: Selecting a row using a clustered index

On the root level page, “Green” is greater than “Bennet,” but less than Karsten, so the pointer for “Bennet” is followed to page 1007. On page 1007, “Green” is greater than “Greene,” but less than “Hunter,” so the pointer to page 1133 is followed to the leaf-level 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**). “Indexes and I/O Statistics” on page 7-9 provides more information on physical and logical I/O and Adaptive Server tools for reporting it. 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. See “Indexes and Caching” on page 4-19 for more details on how indexes use the data cache.

This description covers point queries, queries that use the index key in the **where** clause to find a single row or a small set of rows. Chapter

8, “Understanding the Query Optimizer,” describes processing more complex types of queries.

Clustered Indexes and Insert Operations

When you insert a row into a 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 4-5 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.

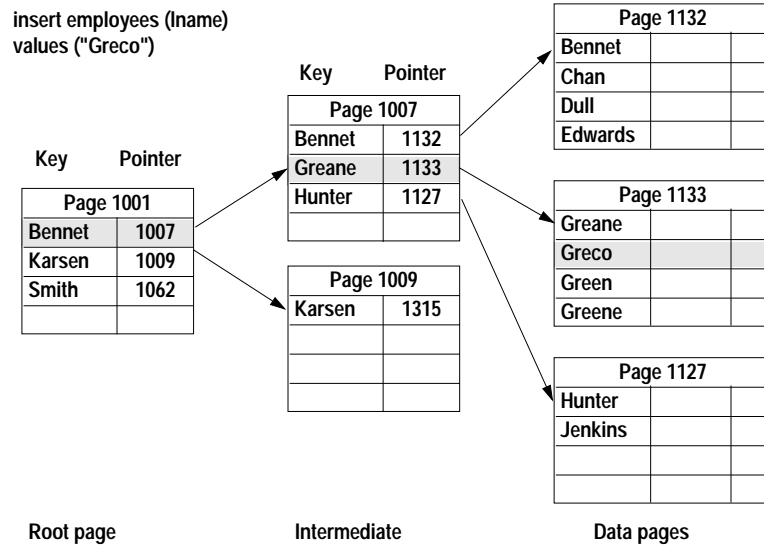


Figure 4-5: Inserting a row into a table with a clustered index

Page Splitting on Full Data Pages

If there is not enough room on the data page for the new row, a page split must be performed:

- A new data page is allocated on an extent already in use by the table. If there is no free page, a new extent is allocated.

- The next and previous page pointers on adjacent pages are changed to incorporate the new page in the page chain. This requires reading those pages into memory and locking them.
- Approximately half of the rows are moved to the new page, with the new row inserted in order.
- The higher levels of the clustered index change to point to the new page.
- If the table also has nonclustered indexes, all of their pointers to the affected data rows must be changed to point to the new page and row locations. See “Nonclustered Indexes” on page 4-12.

In some cases, page splitting is handled slightly differently. See “Exceptions to Page Splitting” on page 4-7.

In Figure 4-6, the page split requires adding a new row to an existing index page, page 1007.

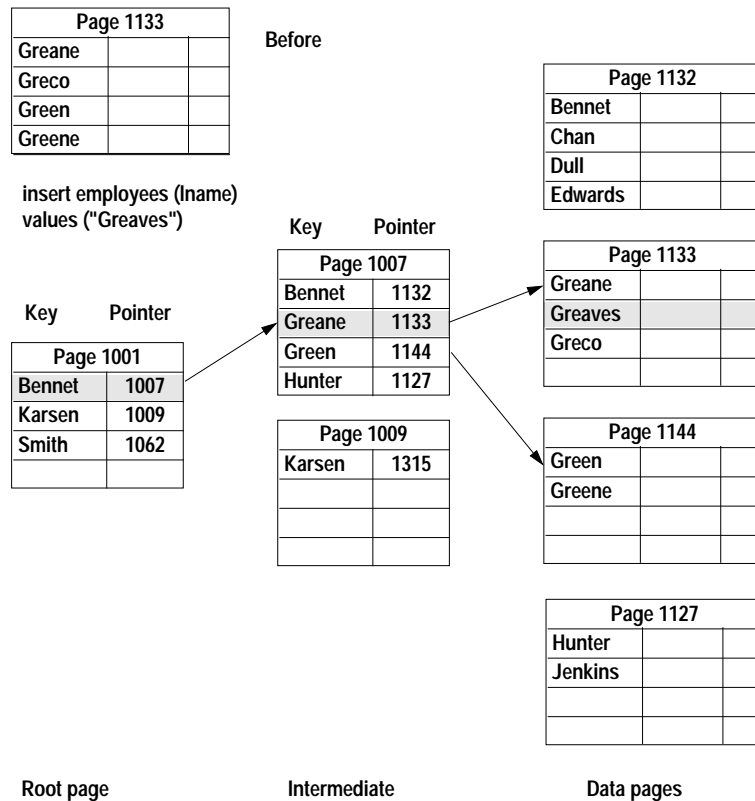


Figure 4-6: Page splitting in a table with a clustered index

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 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 Fillfactors for Indexes” on page 7-48.

Overflow Pages

Special overflow pages are created for nonunique clustered indexes 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 4-7).

insert employees (lname)
values("Greene")

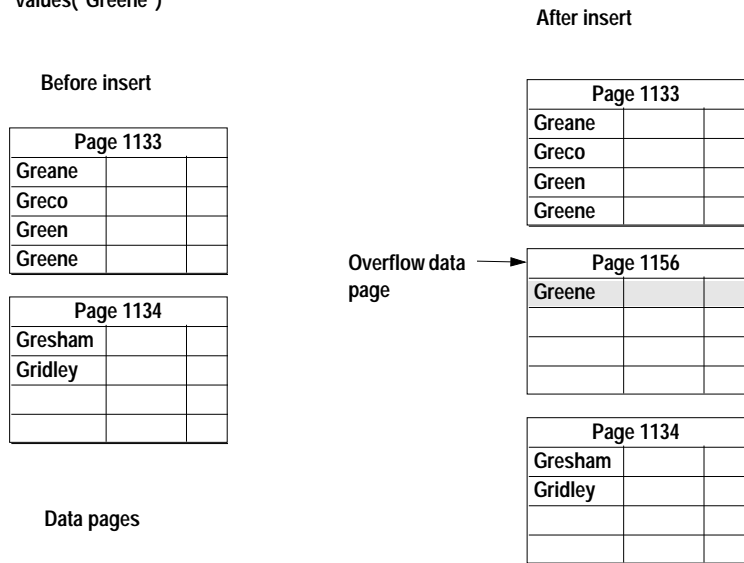


Figure 4-7: Adding an overflow page to a nonunique clustered index

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 a table that has a clustered index, other rows on the page are moved up to fill the empty space so that the

data remains contiguous on the page. Figure 4-8 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.

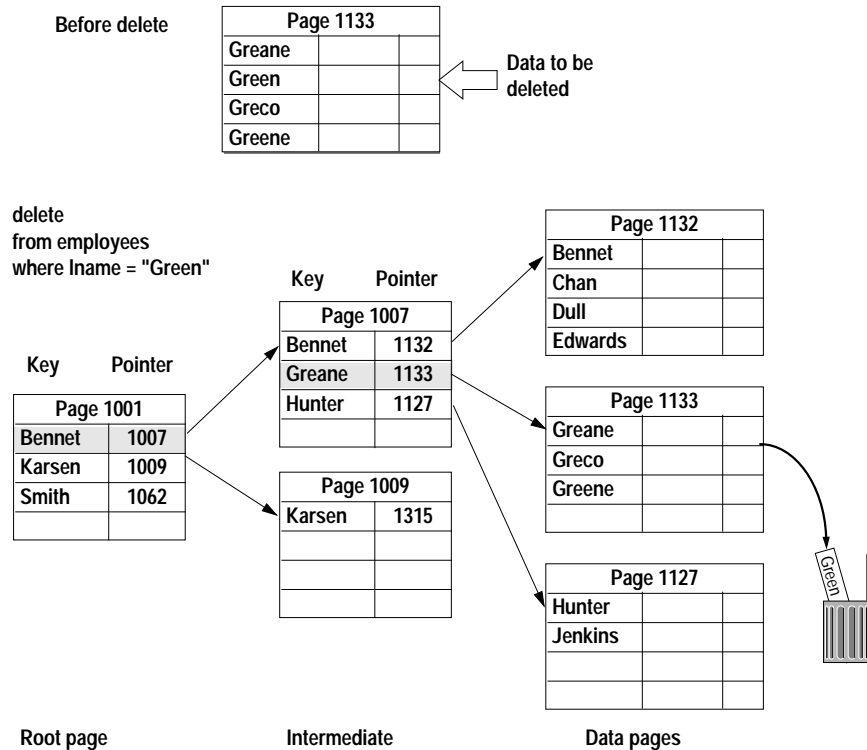


Figure 4-8: Deleting a row from a table with a clustered index

Deleting the Last Row on a Page

If you delete the last row on a data page:

- The page is deallocated
- The next and previous page pointers on the adjacent pages are changed
- The row that points to that page in the intermediate levels of the index is removed

If the deallocated data page is on the same extent as other pages belonging to the table, it is 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.

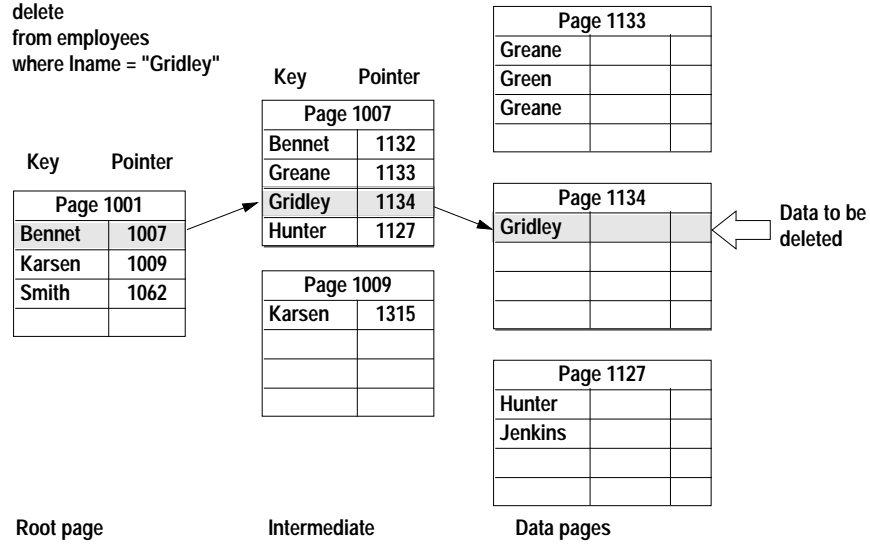


Figure 4-9: Deleting the last row on a page (before the delete)

In Figure 4-10, which shows the table after the delete, 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 space used contiguous.

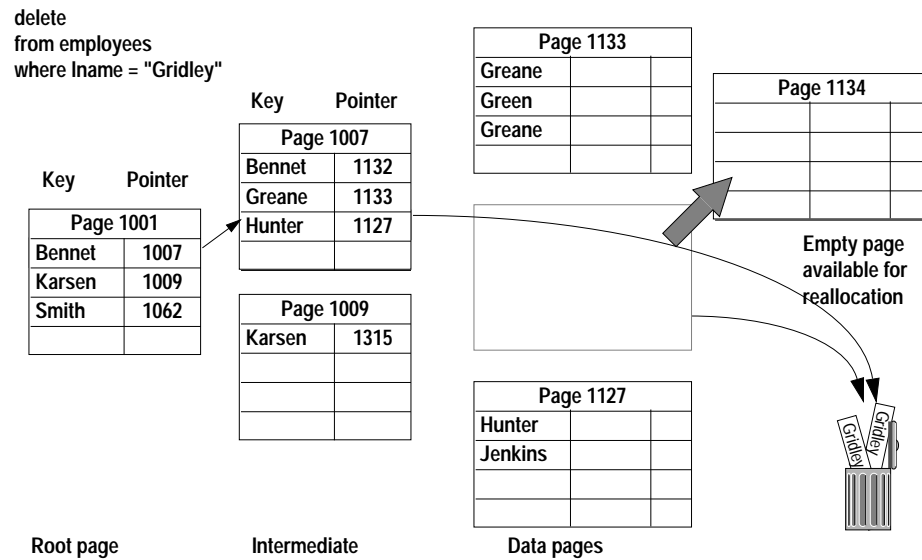


Figure 4-10: Deleting the last row on a page (after the delete)

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

To clearly understand the difference between clustered and nonclustered indexes, it is important to recall the definition of the leaf page of an index: It is the lowest level of the index where all of the keys for the index appear in sorted order.

In clustered indexes, 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 a clustered index that contains one index row for each data row. Clustered indexes are sparse indexes. The level above the data contains one pointer for every data page.

In nonclustered indexes, the level just above the data is the leaf level: it contains a key-pointer pair for each data row. Nonclustered indexes are dense. At the level above the data, they contain one row for each data row.

Row IDs and the Offset Table

Row IDs are managed by an offset table on each data page. The offset table starts at the last byte on the page. There is a 2-byte offset table entry for each row on the page. As rows are added, the offset table grows from the end of the page upward as the rows fill from the top of the page, as shown in Figure 4-11. The offset table stores the byte at which its corresponding row on the page starts.

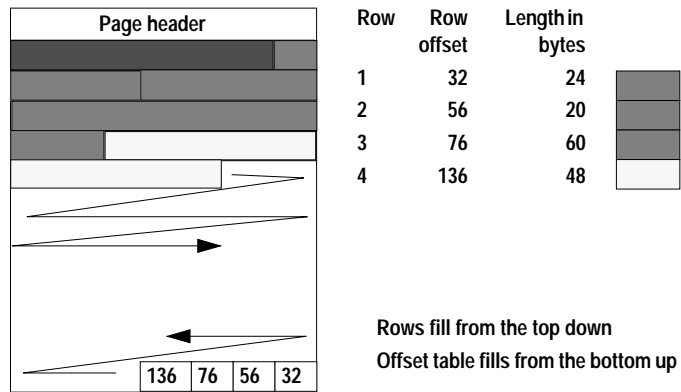


Figure 4-11: Data page with the offset table

When additional rows are inserted between existing rows on the page, an additional value is added to the row offset table, and the offsets for each row are adjusted. The row ID points to the offset table, and the offset table points to the start of the data on the page. When rows are added or deleted, changes are made to the offset table, but the row IDs of the index pointers for existing rows do not change. If you delete a row from the page, its row offset is set to 0.

Figure 4-12 shows the same page that is shown in Figure 4-11 after a new 20-byte row has been inserted as the second row on the page. The existing rows have been moved down and their offset values have been increased. A row offset entry for the new row has been added. Note that the row offset values are not sequential.

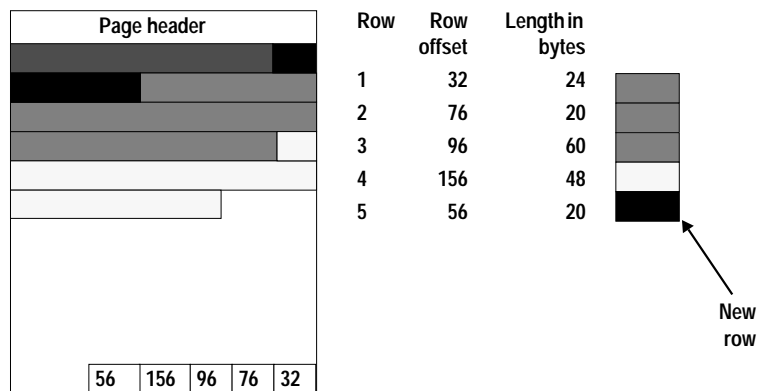


Figure 4-12: Row offset table after an insert

Nonclustered Index Structure

The table in Figure 4-13 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 essential 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.

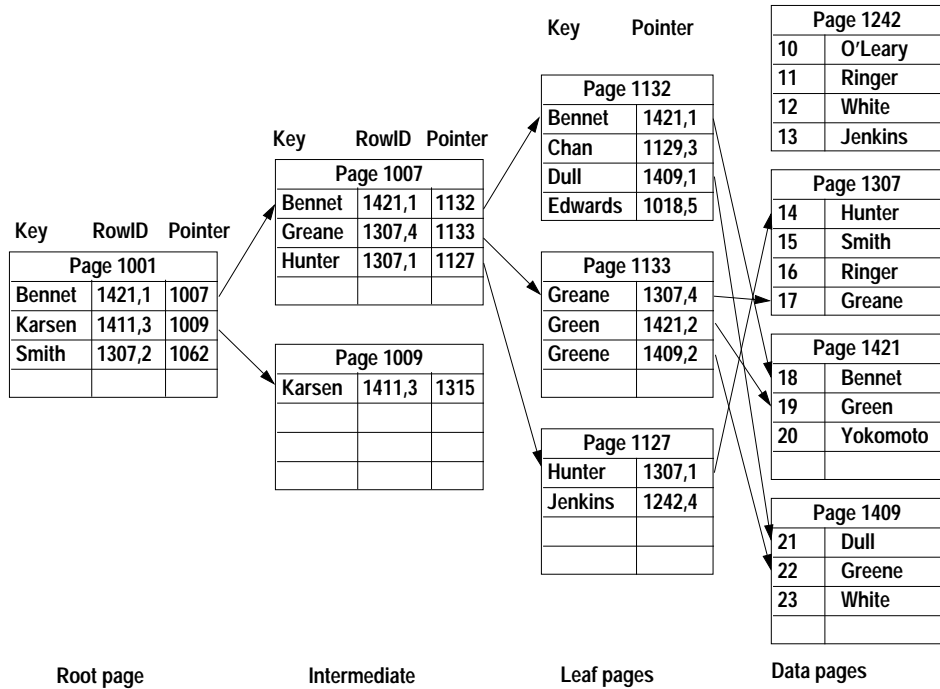


Figure 4-13: Nonclustered index structure

Nonclustered Indexes and Select Operations

When you select a row using a nonclustered index, the search starts at the root level. Just as with clustered indexes, the *sysindexes* table stores the page number for the root page of the nonclustered index.

In Figure 4-14, “Green” is greater than “Bennet,” but less than “Karsen,” so the pointer to page 1007 is followed. “Green” is greater than “Greane,” 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 the second position 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.

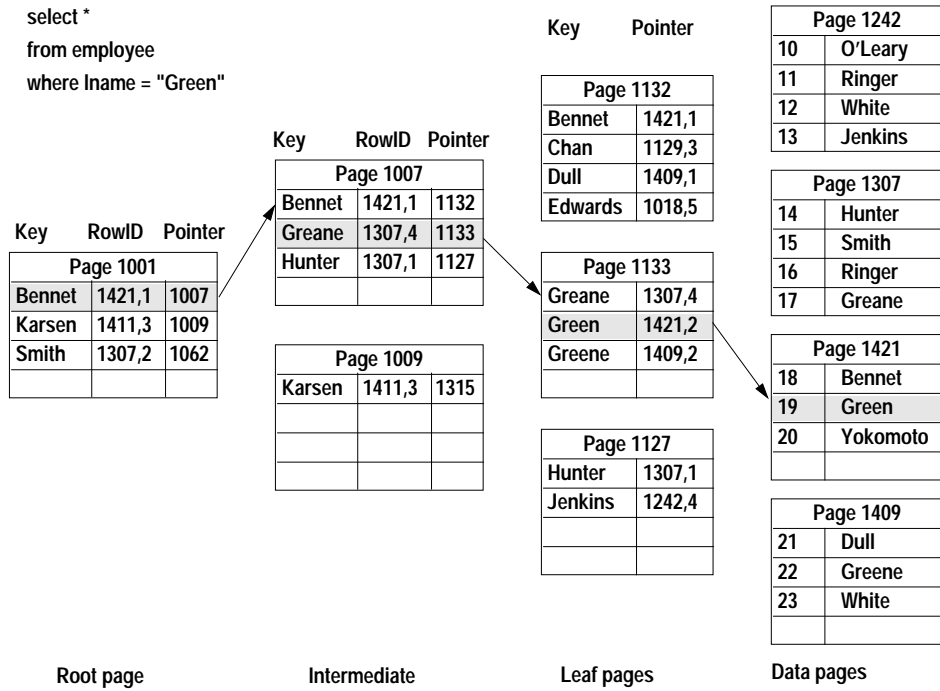


Figure 4-14: Selecting rows using a nonclustered index

Nonclustered Index Performance

The query in Figure 4-14 has 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 actual disk I/Os need to be performed. When Adaptive Server finds a page it needs in the cache, it is called a **logical read**. When Adaptive Server must perform disk I/O, this is called a **physical read**. When a physical read is performed, a logical read is also required.

Nonclustered Indexes and Insert Operations

When you insert rows into a heap that has a nonclustered 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 4-15 shows an insert into a table with a nonclustered index. Since the ID value is 24, 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.

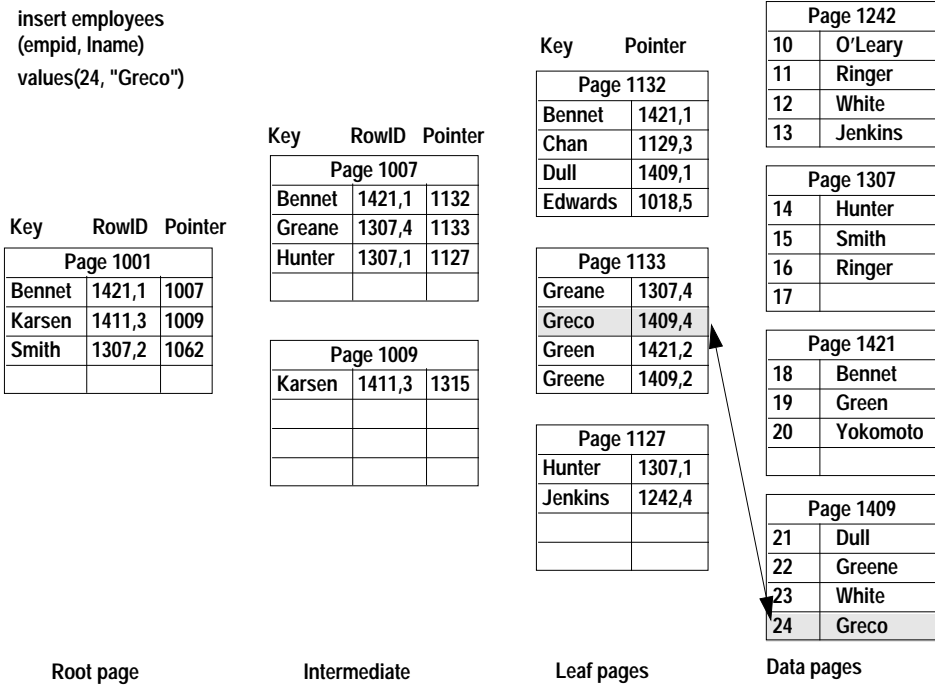


Figure 4-15: An insert with a nonclustered index

Nonclustered Indexes and Delete Operations

When you delete a row from a table, the query can use a nonclustered index on the column or columns in the where clause to locate the data row to delete, as shown in Figure 4-16. 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.

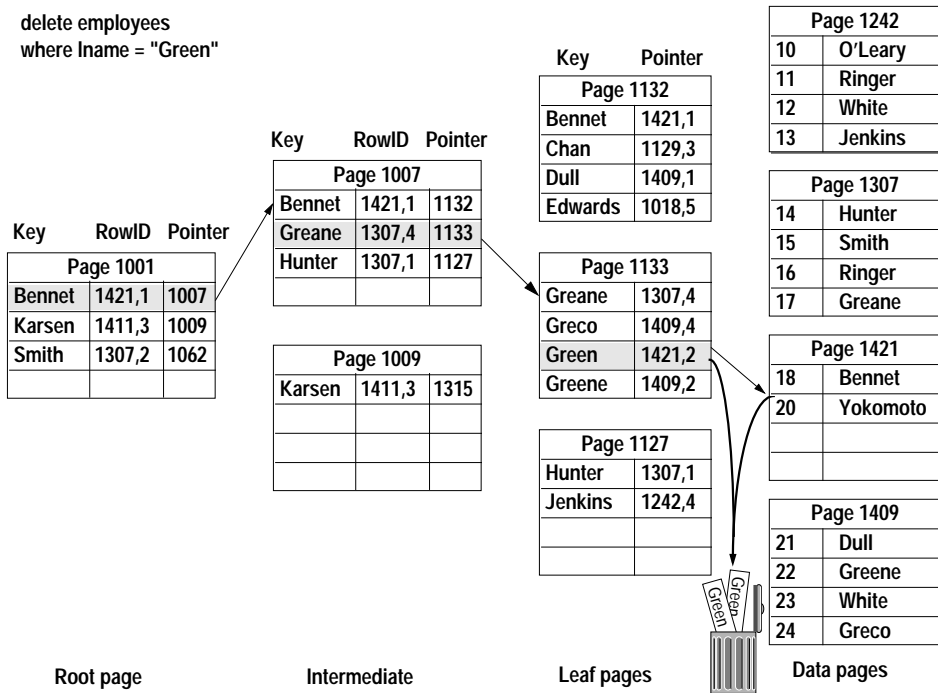


Figure 4-16: Deleting a row from a table with a nonclustered index

If the delete operation removes the last row on the data page, the page is deallocated and the adjacent page pointers are adjusted. 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 4-11.

There is no automatic page merging on data pages, so if your applications make many random deletes, you can end up with data pages that have only a single row, or a few rows, on a page.

Index Covering

Index covering is a nonclustered indexing tactic that can produce dramatic performance improvements. 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 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 data. This is called index covering.

You can create indexes on more than one key. These are called **composite indexes**. Composite indexes can have up to 16 columns adding up to a maximum 600 bytes.

A nonclustered index that covers the query is usually faster than a clustered index, because it reads fewer pages: index rows are smaller, more rows fit on the page, so fewer pages need to be read.

A clustered index, by definition, is covered. Its leaf level contains the complete data rows. This also means that scanning at that level (that is, the entire table) is the same as performing a table scan.

There are two forms of optimization using indexes that cover the query:

- The matching index scan
- The nonmatching index scan

For both types of covered queries, the nonclustered index keys must contain all the columns named in the select list and any clauses of your query: **where**, **having**, **group by**, and **order by**. Matching scans have additional requirements. "Choosing Composite Indexes" on page 7-32 describes query types that make good use of covering indexes.

Matching Index Scans

This type of index covering 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.

In addition to having all columns named in the query included in the index, 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, 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 cannot be used in matching 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. It can use information from the statistics page to estimate the number of pages that need to be read.

In Figure 4-17, 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.

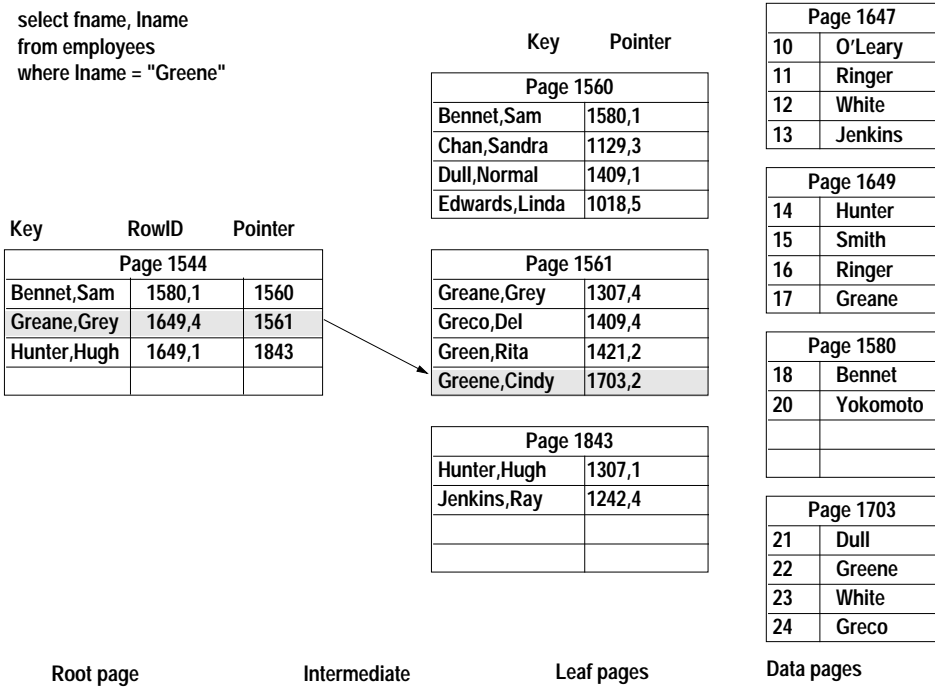


Figure 4-17: Matching index access does not have to read the data row

Nonmatching Index Scans

When the columns specified in the *where* clause do not name 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 leaf level of the nonclustered 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 4-18 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.

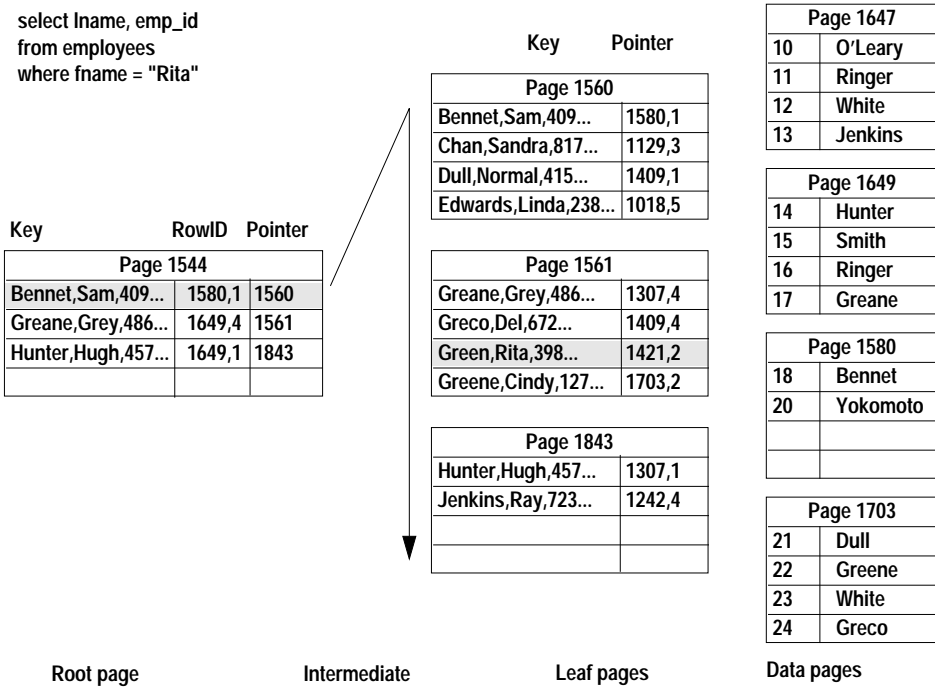


Figure 4-18: A nonmatching index scan

Indexes and Caching

“How Adaptive Server Performs I/O for Heap Operations” on page 3-14 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.
- Nonclustered index scans can use fetch-and-discard strategy.
- Index pages can use one cache while the data pages use a different cache.
- 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. Figure 4-19 shows the data cache just after these 4 pages have been read.

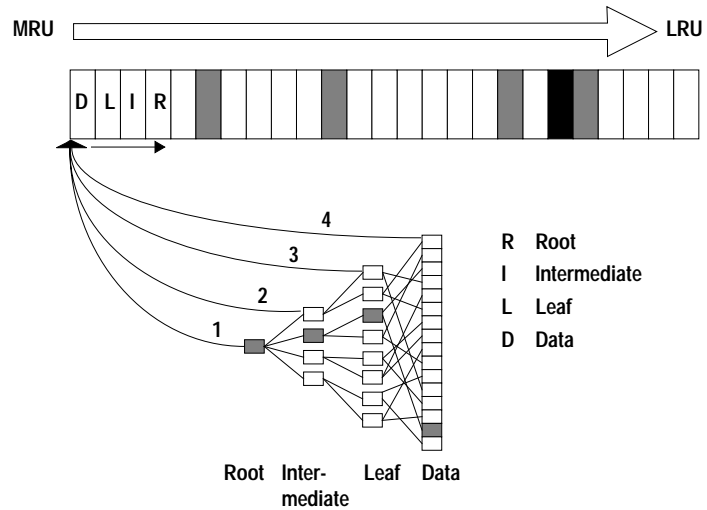


Figure 4-19: Caching used for a point query via a nonclustered index

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. Figure 4-20 shows a root page being moved back to the top of the cache for a second query that uses the same index.

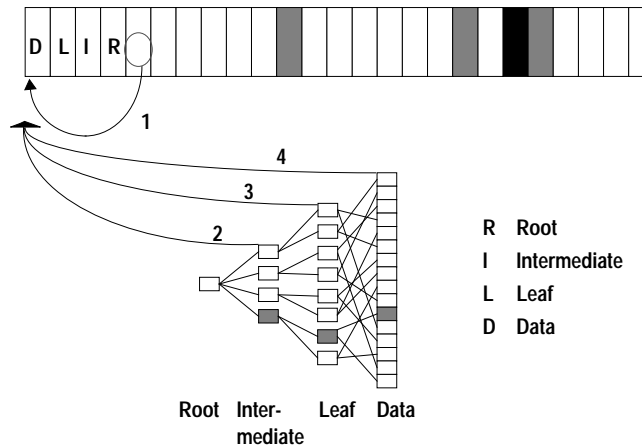


Figure 4-20: Finding the root index page in cache

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. Figure 4-21 shows index pages being read into one cache and a data page into a different cache.

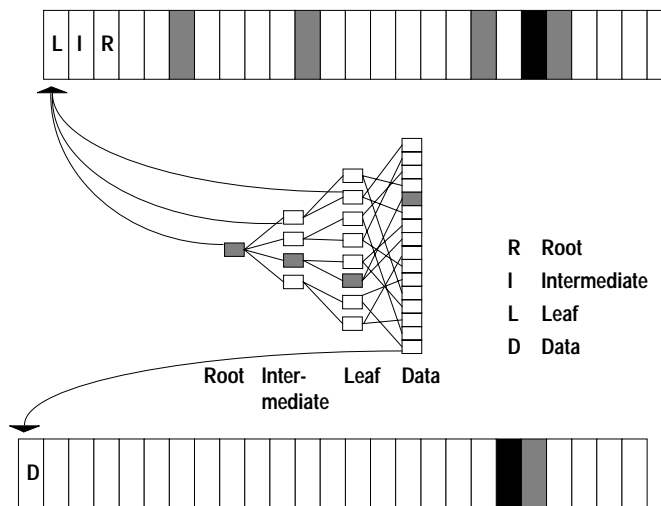


Figure 4-21: Caching with separate caches for data and log

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.

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. Figure 4-22 shows an index page moving from the LRU end of the cache to the MRU end.

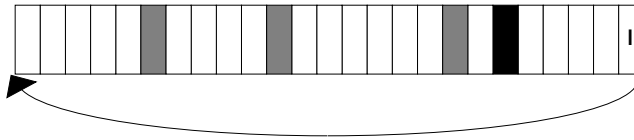


Figure 4-22: Index page recycling in the cache

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 configuration parameter `number of index trips`. For more information, see “number of index trips” on page 11-26 of the *System Administration Guide*.

5

Locking in Adaptive Server

This chapter discusses the types of locks used in Adaptive Server and the commands that can affect locking.

This chapter contains the following sections:

- How Locking Affects Performance 5-1
- Overview of Locking 5-2
- Types of Locks in Adaptive Server 5-3
- How Isolation Levels Affect Locking 5-11
- Controlling Isolation Levels 5-17
- Examples of Locking and Isolation Levels 5-22
- Cursors and Locking 5-24
- Deadlocks and Concurrency 5-26
- Locking and Performance 5-29
- Reporting on Locks and Locking Behavior 5-35
- Configuring Locks and Lock Promotion Thresholds 5-44

How Locking Affects Performance

Adaptive Server protects the tables or data pages currently used by active transactions by locking them. Locking is a concurrency control mechanism: it ensures the consistency of data across transactions. 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.

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 other process's page or table. 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.

Overview of Locking

Consistency of data means that if multiple users repeatedly execute a series of transactions, the results are the same each time. Simultaneous retrievals and modifications of data do not interfere with each other: the results of queries are consistent.

For example, in Figure 5-1, transactions T1 and T2 are attempting to access data at approximately the same time.

T1	Event Sequence	T2
<pre>begin transaction update account set balance = balance - 100 where acct_number = 25 update account set balance = balance + 100 where acct_number = 45 commit transaction</pre>	<p>T1 and T2 start</p> <p>T1 updates balance for one account by subtracting \$100</p> <p>T2 queries the sum balance for several accounts, which is off by \$100 at this point in time—should it return results now, or wait until T1 ends?</p> <p>T1 updates balance of the other account by adding the \$100</p> <p>T1 ends</p>	<pre>begin transaction select sum(balance) from account where acct_number < 50 commit transaction</pre>

Figure 5-1: Consistency levels in transactions

If transaction T2 runs before T1 starts or after T1 completes, both executions of T2 return the same value. But if T2 runs in the middle of transaction T1 (after the first update), the result for transaction T2 will be different by \$100. While such behavior may be acceptable in certain limited situations, most database transactions need to return consistent results.

By default, Adaptive Server locks the data used in T1 until the transaction is finished. Only then does it allow T2 to complete its

query. T2 “sleeps”, or pauses in execution, until the lock it needs is released when T1 is completed.

The alternative, returning data from uncommitted transactions, is known as a **dirty read**. If the results of T2 do not need to be exact, it can read the uncommitted changes from T1, and return results immediately, without waiting for the lock to be released.

Locking is handled automatically by Adaptive Server, with options that can be set at the session and query level by the user. You must know how and when to use transactions to preserve the consistency of your data, while maintaining high performance and throughput. Transactions are described in the *Transact-SQL User's Guide* and in the *Adaptive Server Reference Manual*.

Granularity of Locks

The granularity of locks in a database refers to how much of the data is locked at one time. In theory, a database server can lock as much as the entire database or as little as one row of data. Such extremes affect the concurrency (number of users that can access the data) and locking overhead (amount of work to process each lock request) in the server.

By increasing the lock size, the amount of work required to obtain a lock becomes smaller, but large locks can degrade performance, by making more users wait until locks are released. Decreasing the lock size makes more of the data accessible to other users. However, small locks can also degrade performance, since more work is necessary to maintain and coordinate the increased number of locks. To achieve optimum performance, a locking scheme must balance the needs of concurrency and overhead.

Adaptive Server achieves its balance by locking only data and index pages or entire tables. These locking mechanisms are described in the following sections.

Types of Locks in Adaptive Server

Adaptive Server has two levels of locking: page locks and table locks. Page locks are generally less restrictive (or smaller) than table locks. A page lock locks all the rows on a data page or an index page; a table lock locks an entire table. Adaptive Server uses page locks whenever possible to reduce the contention for data among users and to improve concurrency.

Adaptive Server uses a table lock to provide more efficient locking when it determines that an entire table, or most of its pages, will be accessed by a statement. Locking strategy is directly tied to the query plan, so the query plan can be as important for its locking strategies as for its I/O implications. If an update or delete statement has no useful index, it does a table scan and acquires a table lock. For example, the following statement generates a table lock:

```
update account set balance = balance * 1.05
```

If the update or delete statement uses an index, it begins by acquiring page locks; it tries to acquire a table lock only when a large number of rows are affected.

Whenever possible, Adaptive Server tries to satisfy locking requests with page locks. However, to avoid the overhead of managing hundreds of locks on a table, Adaptive Server uses a **lock promotion threshold** setting. Once a scan of a table accumulates more page locks than allowed by the lock promotion threshold, Adaptive Server tries to issue a table lock. If it succeeds, the page locks are no longer necessary and are released. “Configuring Locks and Lock Promotion Thresholds” on page 5-44 for more information.

Table locks also provide a way to avoid lock collisions at the page level. Adaptive Server automatically uses table locks for some commands.

Adaptive Server makes all locking decisions. It chooses which type of lock to use after it determines the query plan. The way you write a query or transaction can affect the type of lock the server chooses. You can also force the server to make certain locks more or less restrictive by specifying options for select queries or by changing the transaction’s isolation level. See “Controlling Isolation Levels” on page 5-17 for more information.

Page Locks

The following describes the types of page locks:

- **Shared locks**

Adaptive Server applies **shared locks** for read operations. If a shared lock has been applied to a data page or an index page, other transactions can also acquire a shared lock, even when the first transaction is active. However, no transaction can acquire an exclusive lock on the page until all shared locks on the page are released. This means that many transactions can simultaneously read the page, but no transaction can change data or index keys

or pointers on the page while a shared lock exists. Transactions that need an exclusive page lock must wait or “block” for the release of the shared page locks before continuing.

By default, Adaptive Server releases shared page locks after it finishes scanning the page. If Adaptive Server is scanning several pages, it acquires a shared lock on the first page, completes the scan of that page, acquires a lock on the next page, and drops the lock on the first page. It does not hold shared locks until the statement is completed or until the end of the transaction unless special options are specified by the user.

- **Exclusive locks**

Adaptive Server applies **exclusive locks** for data modification operations. When a transaction gets an exclusive lock, other transactions cannot acquire a lock of any kind on the page until the exclusive lock is released at the end of its transaction. The other transactions wait or “block” until the exclusive lock is released.

- **Update locks**

Adaptive Server applies an **update lock** during the initial portion of an update, delete, or fetch (for cursors declared for update) operation while the page is being read. The update lock allows shared locks on the page, but does not allow other update or exclusive locks. Update locks help avoid deadlocks and lock contention. If the page needs to be changed, the update lock is promoted to an exclusive lock as soon as no other shared locks exist on the page.

In general, read operations acquire shared locks, and write operations acquire exclusive locks. For operations that delete or update data, Adaptive Server can apply page-level exclusive and update locks only if the column used in the search argument is part of an index. If no index exists on any of the search arguments, Adaptive Server must acquire a table-level lock.

The following examples show what kind of page locks Adaptive Server uses for the respective statement (with an index on *acct_number*, the search argument used in the queries):

<code>select balance</code>	<i>Shared page lock</i>
<code>from account</code>	
<code>where acct_number = 25</code>	
<code>insert account values</code>	<i>Exclusive page lock on data page</i>
<code>(34, 500)</code>	<i>Exclusive page lock on leaf-level index page</i>
<code>delete account</code>	<i>Update page locks</i>
<code>where balance < 0</code>	<i>Exclusive page locks on data pages</i>
	<i>Exclusive page lock on leaf-level index pages</i>
<code>update account</code>	<i>Update page lock on data page</i>
<code>set balance = 0</code>	<i>Exclusive page lock on data page</i>
<code>where acct_number = 25</code>	

Table Locks

The following describes the types of table locks.

- **Intent lock**

An **intent lock** indicates that certain types of page-level locks are currently held on pages of a table. Adaptive Server applies an intent table lock with each shared or exclusive page lock, so an intent lock can be either an intent exclusive lock or an intent shared lock. Setting an intent lock prevents other transactions from subsequently acquiring conflicting table-level locks on the table that contains that locked page. An intent lock is held as long as the concurrent page locks are in effect.

- **Shared lock**

This lock is similar to the shared page lock, except that it affects the entire table. For example, Adaptive Server applies a shared table lock for a select command with a `holdlock` clause if the command does not use an index. `create nonclustered index` commands also acquire shared table locks.

- **Exclusive lock**

This lock is similar to the exclusive page lock, except that it affects the entire table. For example, Adaptive Server applies an exclusive table lock during a `create clustered index` command. `update` and `delete` statements generate exclusive table locks if their search

arguments do not reference indexed columns of the object (often an indication that you need to examine your indexes or queries.)

The following examples show the respective page and table locks issued for each statement ((with an index on *acct_number*, the search argument used in the queries):

<code>select balance from account where acct_number = 25</code>	<i>Intent shared table lock Shared page lock</i>
<code>insert account values (34, 500)</code>	<i>Intent exclusive table lock Exclusive page lock on data page Exclusive page lock on leaf index pages</i>
<code>delete account where balance < 0</code>	<i>Intent exclusive table lock Update page locks followed by exclusive page locks on data pages and leaf-level index pages</i>

This next example assumes that there is no index on *acct_number*; otherwise, Adaptive Server would attempt to issue page locks for the statement:

<code>update account set balance = 0 where acct_number = 25</code>	<i>Exclusive table lock</i>
--	-----------------------------

Exclusive table locks are also applied to tables during *select into* operations, including temporary tables created with *tempdb.tablename* syntax. Tables created with *#tablename* are restricted to the sole use of the process that created them, and are not locked.

Demand Locks

Adaptive Server sets a **demand lock** to indicate that a transaction is next in the queue to lock a table or page. After a write transaction has been queued and has been skipped over by three tasks or families (in the case of queries running in parallel) acquiring shared locks, Adaptive Server gives a demand lock to the write transaction. Any subsequent requests for shared locks are queued behind the demand lock, as shown in Figure 5-2.

As soon as the readers queued ahead of the demand lock release their locks, the write transaction acquires its lock and is allowed to proceed. The read transactions queued behind the demand lock wait for the write transaction to finish and release its exclusive lock.

Demand locks avoid situations in which read transactions acquire overlapping shared locks, which monopolize a table or page so that a write transaction waits indefinitely for its exclusive lock.

Demand Locking with Serial Execution

Figure 5-2 illustrates how the demand lock scheme works for serial query execution. It shows four tasks with shared locks in the active lock position, meaning that all four tasks are currently reading the page. These tasks can access the same page simultaneously because they hold compatible locks. Two other tasks are in the queue waiting for locks on the page. Here is a series of events that could lead to the situation shown in Figure 5-2:

- Originally, task 2 holds a shared lock on the page.
- Task 6 makes an exclusive lock request, but must wait until the shared lock is released because shared and exclusive locks are not compatible.
- Task 3 makes a shared lock request, which is immediately granted because all shared locks are compatible.
- Tasks 1 and 4 make shared lock requests, which are also immediately granted for the same reason.
- Task 6 has now been skipped 3 times, and is granted a demand lock
- Task 5 makes a shared lock request. It is queued behind task 6's exclusive lock request because task 6 holds a demand lock. Task 5 is the fourth task to make a shared page request.
- After tasks 1, 2, 3, and 4 finish their reads and release their shared locks, task 6 is granted its exclusive lock.
- After task 6 finishes its write and releases its exclusive page lock, task 5 is granted its shared page lock.

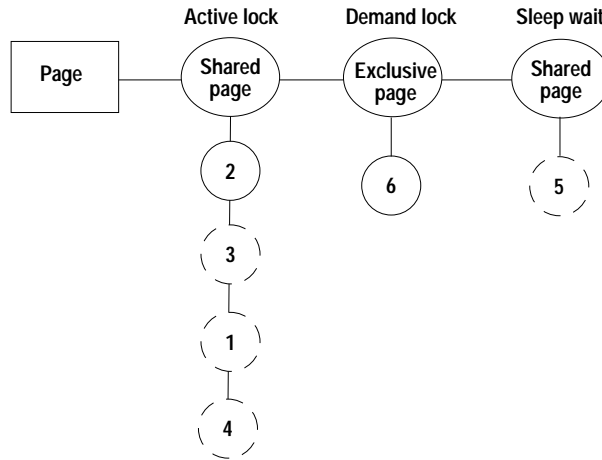


Figure 5-2: Demand locking with serial query execution

Demand Locking with Parallel Execution

When queries are running in parallel, demand locking treats all the shared locks from a family of worker processes as if they were a single task. The demand lock permits the completion of reads from three families (or a total of three tasks and families combined) before granting the exclusive lock. Therefore, a task or worker process in the queue might have to wait much longer to obtain its exclusive lock.

Figure 5-3 illustrates how the demand lock scheme works when parallel query execution is enabled. The figure shows six worker processes from three families with shared locks. A task waits for an exclusive lock, and a worker process from a fourth family waits behind the task. Here is a series of events that could lead to the situation shown in Figure 5-3:

- Originally, worker process 1:3 (worker process 3 from a family with *fid* 1) holds a shared lock on the page.
- Task 9 makes an exclusive lock request, but must wait until the shared lock is released.
- Worker process 2:3 requests a shared lock, which is immediately granted because shared locks are compatible. The skip count for task 9 is now 1.
- Worker processes 1:1, 2:1, 3:1, task 10, and worker processes 3:2 and 1:2 are consecutively granted shared lock requests. Since

family ID 3 and task 10 have no prior locks queued, the skip count for task 9 is now 3, and task 9 is granted a demand lock.

- Finally, worker process 4:1 makes a shared lock request, but it is queued behind task 9's exclusive lock request.
- Any additional shared lock requests from family IDs 1, 2, and 3 and from task 10 are queued ahead of task 9, but all requests from other tasks are queued after it.
- After all the tasks in the active lock position release their shared locks, task 9 is granted its exclusive lock.
- After task 9 releases its exclusive page lock, task 4:1 is granted its shared page lock.

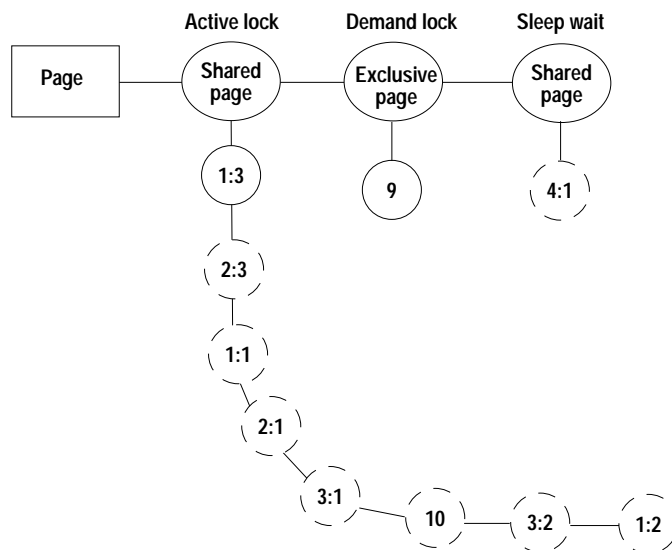


Figure 5-3: Demand locking with parallel query execution

Summary of Lock Types

Table 5-1 describes the types of locks applied by Adaptive Server for insert and create index statements:

Table 5-1: Summary of locks for insert and create index statements

Statement	Table Lock	Data Page Lock
insert	IX	X
create clustered index	X	-
create nonclustered index	S	-
IX = intent exclusive, S = shared, X = exclusive		

Table 5-2 describes the types of locks applied by Adaptive Server for select, delete, and update statements. It divides the select, update, and delete statements into two groups, since the type of locks applied are different if the statement's search argument references indexed columns on the table.

Table 5-2: Summary of locks for select, update and delete statements

Statement	Indexed		Not Indexed	
	Table Lock	Page Lock	Table Lock	Page Lock
select	IS	S	IS	S
select with holdlock	IS	S	S	-
update	IX	U, X	X	-
delete	IX	U, X	X	-
IS = intent shared, IX = intent exclusive, S = shared, U = update, X = exclusive				

Note that the above tables do not describe situations in which Adaptive Server initially uses table locks (if a query requires the entire table) or when it promotes to a table lock after reaching the lock promotion threshold.

Lock Compatibility

Table 5-3 summarizes the information about lock compatibility, showing when locks can be acquired immediately.

Table 5-3: Lock compatibility

If one process has:	Can another process immediately acquire:				
	A Shared Lock?	An Update Lock?	An Exclusive Lock?	A Shared Intent Lock?	An Exclusive Intent Lock?
A Shared Lock	Yes	Yes	No	Yes	No
An Update Lock	Yes	No	No	N/A	N/A
An Exclusive Lock	No	No	No	No	No
A Shared Intent Lock	Yes	N/A	No	Yes	Yes
An Exclusive Intent Lock	No	N/A	No	Yes	Yes

How Isolation Levels Affect Locking

The SQL standard defines four levels of isolation for SQL transactions. Each **isolation level** specifies the kinds of actions that are not permitted while concurrent transactions are executing. Higher levels include the restrictions imposed by the lower levels.

Adaptive Server supports three isolation levels for its transactions: 0, 1, and 3. The requirements for isolation level 2 are included with level 3, but Adaptive Server does not allow you to specifically choose level 2. You can choose which isolation level affects all the transactions executed in your session, or you can choose the isolation level for a specific query in a transaction.

Isolation Level 0

Level 0 prevents other transactions from changing data that has already been modified (using a data modification statement such as **update**) by an uncommitted transaction. The other transactions are blocked from modifying that data until the transaction completes. However, other transactions can still read the uncommitted data—a process known as a **dirty read**. Figure 5-4 shows a dirty read.

T3	Event Sequence	T4
begin transaction	T3 and T4 start	begin transaction
update account set balance = balance - 100 where acct_number = 25	T3 updates balance for one account by subtracting \$100	
	T4 queries current sum of balance for accounts	select sum(balance) from account where acct_number < 50
	T4 ends	commit transaction
rollback transaction	T3 rolls back, invalidating the results from T4	

Figure 5-4: Dirty reads in transactions

If transaction T4 queries the table after T3 updates it, but before it rolls back the change, the amount calculated by T4 is off by \$100.

At isolation level 0, Adaptive Server performs dirty reads by:

- Not applying shared locks on pages or tables being searched.
- Allowing another task to read pages or tables that have exclusive locks. It still applies exclusive locks on pages or tables being changed, which prevents other transactions from changing the data already modified by the uncommitted transaction.

By default, a unique index is required to perform an isolation level 0 read, unless the database is read-only. The index is required to restart the scan if an update by another process changes the query's result set by modifying the current row or page. Forcing the query to use a table scan or a nonunique index can lead to problems, if there is significant update activity on the underlying table, and is not recommended.

Using the example in Figure 5-4, the `update` statement in transaction T3 still acquires an exclusive lock on *account*. The difference with isolation level 0 as opposed to level 1 is that transaction T4 does not try to acquire a shared lock before querying *account*, so it is not blocked by T3. The opposite is also true. If T4 begins to query *accounts* at isolation level 0 before T3 starts, T3 could still acquire its exclusive lock on *accounts* while T4's query executes.

Applications that can use dirty reads may see better concurrency and reduced deadlocks when accessing the same data at a higher isolation level. An example may be transaction T4. If it requires only a snapshot of the current sum of account balances, which probably changes frequently in a very active table, T4 should query the table using isolation level 0. Other applications that require data consistency, such as queries of deposits and withdrawals to specific accounts in the table, should avoid using isolation level 0.

Isolation level 0 can improve performance for applications by reducing lock contention, but can impose performance costs in two ways:

- Dirty reads are implemented by making in-cache copies of dirty data that the isolation level 0 application needs to read.
- If a dirty read is active on a row, and the data changes so that the row is moved or deleted, the scan must be restarted, which may incur additional logical and physical I/O.

The `sp_sysmon` system procedure reports on both of these factors. See “Dirty Read Behavior” on page 24-73.

Even if you set your isolation level to 0, some utilities still acquire shared locks for their scans. Shared locks are necessary for read operations that must maintain the database integrity by ensuring that the correct data is read before modifying it or verifying its consistency.

Isolation Level 1

Level 1 prevents dirty reads. At isolation level 1, if a transaction has modified a row, and a second transaction needs to read that row, the second transaction waits until the first transaction completes (either commits or rolls back.)

For example, compare Figure 5-5, showing a transaction executed at isolation level 1, to Figure 5-4, showing a dirty read transaction.

T5	Event Sequence	T6
<pre>begin transaction update account set balance = balance - 100 where acct_number = 25 rollback transaction</pre>	<p>T5 and T6 start</p> <p>T5 updates account after getting exclusive lock</p> <p>T6 tries to get shared lock to query account but must wait until T5 releases its lock</p> <p>T5 ends and releases its exclusive lock</p> <p>T6 gets shared lock, queries account, and ends</p>	<pre>begin transaction select sum(balance) from account where acct_number < 50 commit transaction</pre>

Figure 5-5: Transaction isolation level 1 prevents dirty reads

When the `update` statement in transaction T5 executes, Adaptive Server applies an exclusive lock (a page-level lock if `acct_number` is indexed; otherwise, a table-level lock) on `account`. The query in T6 cannot execute (preventing the dirty read) until the exclusive lock is released, when T5 ends with the `rollback`.

While the query in T6 holds its shared lock, other processes that need shared locks can access the same data, and an update lock can also be granted (an update lock indicates the read operation that precedes the exclusive-lock write operation), but no exclusive locks are allowed until all shared locks have been released.

Isolation Level 2

Level 2 prevents **nonrepeatable reads**. These occur when one transaction reads a row and a second transaction modifies that row. If the second transaction commits its change, subsequent reads by the first transaction yield results that are different from the original read. Figure 5-6 shows a nonrepeatable read.

T7	Event Sequence	T8
<pre>begin transaction select balance from account where acct_number = 25 select balance from account where acct_number = 25 commit transaction</pre>	<pre>T7 and T8 start T7 queries the balance for one account T8 updates the balance for that same account T8 ends T7 makes same query as before and gets different results T7 ends</pre>	<pre>begin transaction update account set balance = balance - 100 where acct_number = 25 commit transaction</pre>

Figure 5-6: Nonrepeatable reads in transactions

If transaction T8 modifies and commits the changes to the *account* table after the first query in T7, but before the second one, the same two queries in T7 produce different results.

Isolation Level 3

Level 3 prevents **phantoms**. These occur when one transaction reads a set of rows that satisfy a search condition, and then a second transaction modifies the data (through an insert, delete, or update statement). If the first transaction repeats the read with the same search conditions, it obtains a different set of rows. An example of phantoms in transactions is shown in Figure 5-7.

T9	Event Sequence	T10
<pre>begin transaction select * from account where acct_number < 25 select * from account where acct_number < 25 commit transaction</pre>	<p>T9 and T10 start</p> <p>T9 queries a certain set of rows</p> <p>T10 inserts a row that meets the criteria for the query in T9</p> <p>T10 ends</p> <p>T9 makes the same query and gets a new row</p> <p>T9 ends</p>	<pre>begin transaction insert into account (acct_number, balance) values (19, 500) commit transaction</pre>

Figure 5-7: Phantoms in transactions

If transaction T10 inserts rows into the table that satisfy T9's search condition after the T9 executes the first select, subsequent reads by T9 using the same query result in a different set of rows.

Adaptive Server also prevents nonrepeatable reads (the restrictions imposed by level 2 are included in level 3) and phantoms by:

- Applying exclusive locks on pages or tables being changed. It holds those locks until the end of the transaction.
- Applying shared locks on pages or tables being searched. It holds those locks until the end of the transaction.

Using and holding the exclusive and shared locks allows Adaptive Server to maintain the consistency of the results at isolation level 3. However, holding the shared lock until the transaction ends decreases Adaptive Server's concurrency by preventing other transactions from getting their exclusive locks on the data.

Compare the phantom, shown in Figure 5-7, with the same transaction executed at isolation level 3, as shown in Figure 5-8.

T11	Event Sequence	T12
<pre>begin transaction select * from account holdlock where acct_number < 25 select * from account holdlock where acct_number < 25 commit transaction</pre>	<p>T11 and T12 start</p> <p>T11 queries account and holds acquired shared locks</p> <p>T12 tries to insert row but must wait until T11 releases its locks</p> <p>T11 makes same query and gets same results</p> <p>T11 ends and releases its shared locks</p> <p>T12 gets its exclusive lock, inserts new row, and ends</p>	<pre>begin transaction insert into account (acct_number, balance) values (19, 500) commit transaction</pre>

Figure 5-8: Avoiding phantoms in transactions

In transaction T11, Adaptive Server applies shared page locks (if an index exists on the *acct_number* argument) or a shared table lock (if no index exists) and holds those locks until the end of T11. The insert in T12 cannot get its exclusive lock until T11 releases those shared locks. If T11 is a long transaction, T12 (and other transactions) may wait for longer periods of time using isolation level 3 instead of the other levels. As a result, you should use level 3 only when required.

Adaptive Server Default Isolation Level

Adaptive Server's default isolation level is 1, which prevents dirty reads. Adaptive Server enforces isolation level 1 by:

- Applying exclusive locks on pages or tables being changed. It holds those locks until the end of the transaction. Only a process at isolation level 0 can read a page locked by an exclusive lock.
- Applying shared locks on pages being searched. It releases those locks after processing the page or table.

Using the exclusive and shared locks allows Adaptive Server to maintain the consistency of the results at isolation level 1. Releasing the shared lock after the scan moves off a page improves Adaptive Server's concurrency by allowing other transactions to get their exclusive locks on the data.

Controlling Isolation Levels

Adaptive Server's default transaction isolation level is 1. You can change the locking behavior for sessions or individual queries to make locks more or less restrictive.

- Using `set transaction isolation level` to set the level for an entire session to level 0, 1 or 3.
- Specifying the isolation level in the `select` and `readtext` commands using the `at isolation level` clause. The choices are: `read uncommitted`, `read committed`, and `serializable`.
- Using the `holdlock`, `noholdlock`, or `shared` option of the `select` command to specify the isolation level for a single query.

When choosing locking levels in your applications, use the minimum locking level that is consistent with your business model. The combination of setting the session level while providing control over locking behavior at the query level allows concurrent transactions to achieve the results that are required with the least blocking.

Syntax for Query Level Locking Options

The `holdlock`, `noholdlock`, and `shared` options can be specified for each table in a `select` statement, with the `at isolation` clause applies to the entire query.

```
select select_list
from table_name [holdlock | noholdlock] [shared]
  [, table_name [holdlock | noholdlock] [shared]
  {where/group by/order by/compute clauses}
  [at isolation {read uncommitted | read committed |
  serializable}]
```

Here is the syntax for the `readtext` command:

```

readtext table_name.column_name
text_pointer offset size [holdlock]
[using {bytes | chars | characters}]
[at isolation {read uncommitted | read committed |
serializable}]

```

Setting Isolation Levels for a Session

The SQL standard specifies a default isolation level of 3. To enforce this level, Transact-SQL provides the `set transaction isolation level` command. For example, you can make level 3 the default isolation level for your session as follows:

```
set transaction isolation level 3
```

If the session has enforced isolation level 3, you can make the query operate at level 1 using `noholdlock`, as described below.

If you are using the default isolation level of 1, or if you have used the `set transaction isolation level` command to specify level 0, you can enforce level 3 by using the `holdlock` option to hold shared locks until the end of a transaction.

The current isolation level for a session can be determined with the global variable `@@isolation`.

Using *holdlock*, *noholdlock*, or *shared*

You can override a session's locking level with the `holdlock`, `noholdlock`, and `shared` options of the `select` and `readtext` commands:

Level	Keyword	Effect
1	<code>noholdlock</code>	Do not hold locks until the end of the transaction; use from level 3 to enforce level 1
3	<code>holdlock</code>	Hold shared locks until the transaction completes; use from level 1 to enforce level 3
N/A	<code>shared</code>	Applies shared rather than update locks in cursors open for update

These keywords affect locking for the transaction: if you use `holdlock`, the all locks are held until the end of the transaction.

If you specify `holdlock` and isolation level 0 in a query, Adaptive Server issues a warning and ignores the `at isolation` clause.

Using the *at isolation* Clause

You can also change the isolation level for a query by using the *at isolation* clause with a *select* or *readtext* command. The options in the *at isolation* clause are:

Level	Option	Effect
0	<i>read uncommitted</i>	Reads uncommitted changes; use from level 1 or 3 to perform dirty reads (level 0)
1	<i>read committed</i>	Reads only committed changes; wait for locks to be released; use from level 0 to read only committed changes, but without holding locks
3	<i>serializable</i>	Holds shared locks until the query completes; use from level 0 enforce level 1

For example, the following statement queries the *titles* table at isolation level 0:

```
select *
from titles
at isolation read uncommitted
```

For more information about the *transaction isolation level* option and the *at isolation* clause, see Chapter 18, “Transactions: Maintaining Data Consistency and Recovery,” in the *Transact-SQL User’s Guide*.

Making Locks More Restrictive

If isolation level 1 is sufficient for most of your work, but certain queries require higher levels of isolation, you can selectively enforce the highest isolation level using the *holdlock* keyword or *at isolation serializable* in a *select* statement. The *holdlock* keyword makes a shared page or table lock more restrictive. It applies:

- To shared locks
- To the table or view for which it is specified
- For the duration of the statement or transaction containing the statement

The *at isolation serializable* clause applies to all tables in the *from* clause, and is applied only for the duration of the transaction. The locks are released when the transaction completes.

In a transaction, **holdlock** instructs Adaptive Server to hold shared locks until the completion of that transaction instead of releasing the lock as soon as the required table, view, or data page is no longer needed. Adaptive Server always holds exclusive locks until the end of a transaction.

When you use **holdlock** or **serializable**, locking depends on how the data is accessed, as shown in Table 5-4.

Table 5-4: Access method and lock types

Access method	Type of locks
Table scan	Shared table lock
Clustered index	Intent shared table lock; Shared lock on data pages by the query
Nonclustered index	Intent shared table lock; Shared lock on data pages and on nonclustered leaf-level pages used by the query
Covering nonclustered index	Intent shared table lock; Shared lock on nonclustered leaf pages used by the query

The use of **holdlock** in the following example ensures that the two queries return consistent results:

```
begin transaction
select branch, sum(balance)
  from account holdlock
  group by branch
select sum(balance) from account
commit transaction
```

The first query acquires a shared table lock on *account* so that no other transaction can update the data before the second query runs. This lock is not released until the transaction including the **holdlock** command completes.

Using *read committed*

If your session isolation level is 0, and you need to read only committed changes to the database, you can use the **at isolation level read committed** clause.

Making Locks Less Restrictive

In contrast to `holdlock`, the `noholdlock` keyword prevents Adaptive Server from holding any shared locks acquired during the execution of the query, regardless of the transaction isolation level currently in effect. `noholdlock` is useful in situations where your transactions require a default isolation level of 3. If any queries in those transactions do not need to hold shared locks until the end of the transaction, you should specify `noholdlock` with those queries to improve the concurrency of your server.

For example, if your transaction isolation level is set to 3, which would normally cause a `select` query to hold locks until the end of the transaction, this command does releases the locks when the scan moves off the page:

```
select balance from account noholdlock
      where acct_number < 100
```

Using *read uncommitted*

If your session isolation level is 1 or 3, and you want to perform dirty reads, you can use the `at isolation level read uncommitted` clause.

Using *shared*

The `shared` keyword instructs Adaptive Server to use a shared lock (instead of an update lock) on a specified table or view in a cursor. See “Using the shared Keyword” on page 5-25 for more information.

Examples of Locking and Isolation Levels

This section describes the sequence of locks applied by Adaptive Server for the two transactions in Figure 5-9.

T13	Event Sequence	T14
<pre>begin transaction update account set balance = balance - 100 where acct_number = 25 update account set balance = balance + 100 where acct_number = 45 commit transaction</pre>	<p>T13 and T14 start</p> <p>T13 gets exclusive lock and updates account</p> <p>T14 tries to query account but must wait until T13 ends</p> <p>T13 keeps updating account and gets more exclusive locks</p> <p>T13 ends and releases its exclusive locks</p> <p>T14 gets shared locks, queries account, and ends</p>	<pre>begin transaction select sum(balance) from account where acct_number < 50 commit transaction</pre>

Figure 5-9: Locking example between two transactions

The following sequence of locks assumes that an index exists on the *acct_number* column in the *account* table, a default isolation level of 1, and 10 rows per page (50 rows divided by 10 equals 5 data pages):

T13 Locks	T14 Locks
<pre>Intent exclusive table lock on account Update lock page 1 Exclusive lock page 1 Update lock page 5 Exclusive lock page 5 Release all locks at commit</pre>	<pre>Intent shared table lock on account denied, wait for release Intent shared table lock on account granted Shared lock page 1 Shared lock page 2, release lock page 1 Shared lock page 3, release lock page 2 Shared lock page 4, release lock page 3 Shared lock page 5, release lock page 4 Release lock, page 5 Release intent shared table lock</pre>

If no index exists for *acct_number*, Adaptive Server applies exclusive table locks instead of page locks for T13:

T13 Locks	T14 Locks
Exclusive table lock on account Release exclusive table lock at commit	Intent shared table lock on account denied, wait for release Intent shared table lock on account granted Shared lock page 1 Shared lock page 2, release lock page 1 Shared lock page 3, release lock page 2 Shared lock page 4, release lock page 3 Shared lock page 5, release lock page 4 Release lock page 5 Release intent shared table lock

If you add a **holdlock** or set the transaction isolation level to 3 for transaction T14, the lock sequence is as follows (assuming an index exists for *acct_number*):

T13 Locks	T14 Locks
Intent exclusive table lock on account Update lock page 1 Exclusive lock page 1 Update lock page 5 Exclusive lock page 5 Release all locks at commit	Intent shared table lock on account 1 denied, wait for release Intent shared table lock on account 1 granted Shared lock page 1 Shared lock page 2 Shared lock page 3 Shared lock page 4 Shared lock page 5 Release all locks at commit

If you add **holdlock** or set the transaction isolation level to 3 for T14, and no index exists for *acct_number*, Adaptive Server applies table locks for both transactions instead of page locks:

T13 Locks	T14 Locks
Exclusive table lock on account Release exclusive table lock at commit	Shared table lock denied, wait for release Shared table lock on account Release shared table lock at commit

Cursors and Locking

Cursor locking methods are similar to the other locking methods in Adaptive Server. For cursors declared as **read only** or declared without the **for update** clause, Adaptive Server uses a shared page lock on the data page that includes the current cursor position. When additional rows for the cursor are fetched, Adaptive Server acquires a lock on the next page, the cursor position is moved to that page, and the previous page lock is released (unless you are operating at isolation level 3.)

For cursors declared with **for update**, Adaptive Server uses update page locks by default when scanning tables or views referenced with the **for update** clause of the cursor. If the **for update** list is empty, all tables and views referenced in the **from** clause of the *select_statement* receive update locks. An update lock is a special type of read lock that indicates that the reader may modify the data soon. An update lock allows other shared locks on the page, but does not allow other update or exclusive locks.

If a row is updated or deleted through a cursor, the data modification transaction acquires an exclusive lock. Any exclusive locks acquired by updates through a cursor in a transaction are held until the end of that transaction and are not affected by closing the cursor. This is also true of shared or update locks for cursors that use the **holdlock** keyword or isolation level 3.

The following describes the locking behavior for cursors at each isolation level:

- At level 0, Adaptive Server uses no locks on any base table page that contains a row representing a current cursor position. Cursors acquire no read locks for their scans, so they do not block other applications from accessing the same data. However,

cursors operating at this isolation level are not updatable, and they require a unique index on the base table to ensure accuracy.

- At level 1, Adaptive Server uses shared or update locks on base table or leaf-level index pages that contain a row representing a current cursor position. The page remains locked until the current cursor position moves off the page as a result of fetch statements.
- At level 3, Adaptive Server uses shared or update locks on any base table or leaf-level index pages that have been read in a transaction through the cursor. Adaptive Server holds the locks until the transaction ends; it does not release the locks when the data page is no longer needed or when the cursor is closed.

If you do not set the `close on endtran` or `chained` options, a cursor remains open past the end of the transaction, and its current page locks remain in effect. It could also continue to acquire locks as it fetches additional rows.

Using the *shared* Keyword

When declaring an updatable cursor using the `for update` clause, you can tell Adaptive Server to use shared page locks (instead of update page locks) in the cursor's `declare cursor` statement:

```
declare cursor_name cursor
  for select select_list
  from {table_name | view_name} shared
  for update [of column_name_list]
```

This allows other users to obtain an update lock on the table or an underlying table of the view. You can use `shared` only with the `declare cursor` statement.

You can use the `holdlock` keyword in conjunction with `shared` after each table or view name, but `holdlock` must precede `shared` in the `select` statement. For example:

```
declare authors_crshr cursor
  for select au_id, au_lname, au_fname
  from authors holdlock shared
  where state != 'CA'
  for update of au_lname, au_fname
```

These are the effects of specifying the `holdlock` or `shared` options (of the `select` statement) when defining an updatable cursor:

- If you do not specify either option, the cursor holds an update lock on the page containing the current row. Other users cannot update, through a cursor or otherwise, a row on this page. Other users can declare a cursor on the same tables you use for your cursor, but they cannot get a shared or update lock on your current page.
- If you specify the `shared` option, the cursor holds shared locks on the page containing the currently fetched row. Other users cannot update, through a cursor or otherwise, the rows on this page. They can, however, read rows on the page.
- If you specify the `holdlock` option, you hold update locks on all the pages that have been fetched (if transactions are not being used) or only the pages fetched since the last commit or rollback (if in a transaction). Other users cannot update, through a cursor or otherwise, currently fetched pages. Other users can declare a cursor on the same tables you use for your cursor, but they cannot get an update lock on currently fetched pages.
- If you specify both options, the cursor holds shared locks on all the pages fetched (if not using transactions) or on the pages fetched since the last commit or rollback. Other users cannot update, through a cursor or otherwise, currently fetched pages.

Deadlocks and Concurrency

Simply stated, a **deadlock** occurs when two user processes each have a lock on a separate data page, index page, or table and each wants to acquire a lock on the other process's page or table. When this happens, the first process is waiting for the second to let go of the lock, but the second process will not let it go until the lock on the first process's object is released. Figure 5-10 shows an example of a deadlock between two processes (a deadlock often involves more than two processes).

T15	Event Sequence	T16
begin transaction	T15 and T16 start	begin transaction
update savings set balance = balance - 250 where acct_number = 25	T15 gets exclusive lock for savings while T16 gets exclusive lock for checking	update checking set balance = balance - 75 where acct_number = 45
update checking set balance = balance + 250 where acct_number = 45	T15 waits for T16 to release its lock while T16 waits for T15 to release its lock; deadlock occurs	update savings set balance = balance + 75 where acct_number = 25
commit transaction		commit transaction

Figure 5-10: Deadlocks in transactions

Table 5-11 shows the deadlock graphically. See Figure 5-12 on page 5-42 for an illustration of a deadlock involving parallel processes.

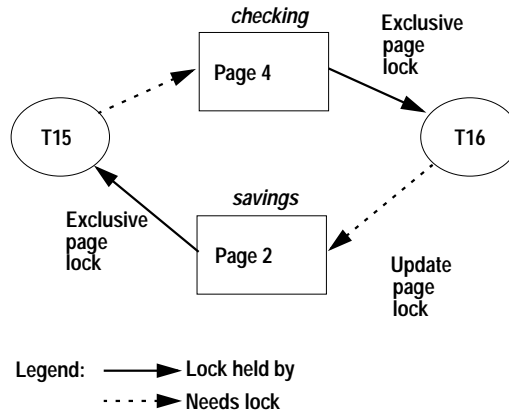


Figure 5-11: A deadlock between two processes

If transactions T15 and T16 execute simultaneously, and both transactions acquire exclusive locks with their initial update statements, they deadlock, waiting for each other to release their locks, which will not happen.

Adaptive Server checks for deadlocks and chooses the user whose transaction has accumulated the least amount of CPU time as the

victim. Adaptive Server rolls back that user's transaction, notifies the application program of this action with message number 1205, and allows the other user processes to move forward.

In a multiuser situation, each user's application program should check every transaction that modifies data for message 1205 if there is any chance of deadlocking. It indicates that the user transaction was selected as the victim of a deadlock and rolled back. The application program must restart that transaction.

Avoiding Deadlocks

It is possible to encounter deadlocks when many long-running transactions are executed at the same time in the same database. Deadlocks become more common as the lock contention increases between those transactions, which decreases concurrency. Methods for reducing lock contention, such as avoiding table locks and not holding shared locks, are described in "Locking and Performance" on page 5-29.

Acquire Locks on Objects in the Same Order

Well-designed applications can minimize deadlocks by always acquiring locks in the same order. Updates to multiple tables should always be performed in the same order.

For example, the transactions described in Figure 5-10 could have avoided their deadlock by updating either the *savings* or *checking* table first in both transactions. That way, one transaction gets the exclusive lock first and proceeds while the other transaction waits to receive its exclusive lock on the same table when the first transaction ends.

In applications with large numbers of tables and transactions that update several tables, establish a locking order that can be shared by all application developers.

Adaptive Server also avoids deadlocks by using the following locks:

- Update page locks permit are a special type of shared lock that both increases concurrency and reduces deadlocks. Shared locks are compatible with update locks, so they allow concurrent read access. The update lock is held while the row is examined to determine if it needs to be updated. If so, the lock is promoted to an exclusive lock. Trying to allow this concurrency by holding

regular shared locks, and then changing them to exclusive locks could actually increase deadlocking.

- Intent table locks act as a placeholder when shared or exclusive page locks are acquired. They inform other transactions that need a lock whether or not the lock can be granted without having Adaptive Server scan the page lock queue. They also help avoid lock collisions between page level locks and table level locks.

Delaying Deadlock Checking

Adaptive Server performs deadlock checking after a minimum period of time for any process waiting for a lock to be released (sleeping). This deadlock checking is time-consuming overhead for applications that wait without a deadlock.

If your applications deadlock infrequently, Adaptive Server can delay deadlock checking and reduce the overhead cost. You can specify the minimum amount of time (in milliseconds) that a process waits before it initiates a deadlock check using the configuration parameter `deadlock checking period`. Valid values are 0–2147483. The default value is 500. `deadlock checking period` is a dynamic configuration value, so any change to it takes immediate effect.

If you set the value to 0, Adaptive Server initiates deadlock checking when the process begins to wait for a lock. If you set the value to 600, Adaptive Server initiates a deadlock check for the waiting process after at least 600 ms. For example:

```
sp_configure "deadlock checking period", 600
```

Setting `deadlock checking period` to a higher value produces longer delays before deadlocks are detected. However, since Adaptive Server grants most lock requests before this time elapses, the deadlock checking overhead is avoided for those lock requests. If you expect your applications to deadlock infrequently, you can set `deadlock checking period` to a higher value and avoid the overhead of deadlock checking for most processes.

Adaptive Server performs deadlock checking for all processes at fixed intervals, determined by `deadlock checking period`. If Adaptive Server performs a deadlock check while a process's deadlock checking is delayed, the process waits until the next interval. Therefore, a process may wait from the number of milliseconds set by `deadlock checking period` to almost twice that value before deadlock checking is performed. Information from `sp_sysmon` can help you tune deadlock checking behavior. See "Deadlock Detection" on page 24-63.

Locking and Performance

Locking affects performance of Adaptive Server by limiting concurrency. An increase in the number of simultaneous users of a server may increase lock contention, which decreases performance. Locks affect performance when:

- Processes wait for locks to be released

Anytime a process waits for another process to complete its transaction and release its locks, the overall response time and throughput is affected.

- Transactions result in frequent deadlocks

As described earlier, any deadlock causes one transaction to be aborted, and the transaction must be restarted by the application. If deadlocks occur often, it severely affects the throughput of applications. Deadlocks cannot be completely avoided. However, redesigning the way transactions access the data can help reduce their frequency.

- Creating indexes locks tables

Creating a clustered index locks all users out of the table until the index is created. Creating a nonclustered index locks out all updates until it is created. Either way, you should create indexes at a time when there is little activity on your server.

- Turning off delayed deadlock detection causes spinlock contention.

Setting the deadlock checking period to 0 causes more frequent deadlock checking. The deadlock detection process holds spinlocks on the lock structures in memory while it looks for deadlocks. In a high transaction production environment, do not set this parameter to 0.

Using *sp_sysmon* While Reducing Lock Contention

Many of the following sections suggest changing configuration parameters to reduce lock contention. Use *sp_sysmon* to determine if lock contention is a problem, and then use it to determine how tuning to reduce lock contention affects the system. See “Lock Management” on page 24-58 for more information about using *sp_sysmon* to view lock contention.

If lock contention is a problem, you can use Adaptive Server Monitor to pinpoint locking problems by checking locks per object.

Reducing Lock Contention

Lock contention can have a large impact on Adaptive Server's throughput and response time. You need to consider locking during database design, and monitor locking during application design. Redesigning the tables that have the highest lock contention may improve performance.

For example, an `update` or `delete` statement that has no useful index on its search arguments performs a table scan and holds a table lock for the entire scan time. Table locks generate more lock contention than page locks, since no other process can access the table. Creating a useful index for the query allows the data modification statement to use page locks, improving concurrent access to the table.

If creating an index for a lengthy update or delete transaction is not possible, you can perform the operation in a cursor, with frequent commit transaction statements to reduce the number of page locks.

Keeping Transactions Short

Any transaction that acquires locks should be kept as short as possible. In particular, avoid transactions that need to wait for user interaction while holding locks.

<code>begin tran</code>	
<code>select balance</code>	<i>Intent shared table lock</i>
<code>from account holdlock</code>	<i>Shared page lock</i>
<code>where acct_number = 25</code>	If the user goes to lunch now, no one can update rows on the page that holds this row
<code>update account</code>	<i>Intent exclusive table lock</i>
<code>set balance = balance + 50</code>	<i>Update page lock on data page</i>
<code>where acct_number = 25</code>	<i>Exclusive page lock on data page</i>
	If the user goes to lunch now, no one can read rows on the page that holds this row
<code>commit tran</code>	

Avoid network traffic as much as possible within transactions. The network is slower than Adaptive Server. The example below shows a transaction executed from isql, sent as two packets.

```

begin tran                                isql batch sent to Adaptive Server
update account                             Locks held waiting for commit
set balance = balance + 50
where acct_number = 25
go

update account                             isql batch sent to Adaptive Server
set balance = balance - 50                 Locks released
where acct_number = 45
commit tran
go
    
```

Keeping transactions short is especially crucial for data modifications that affect nonclustered index keys. Nonclustered indexes are dense: the level above the data level contains one row for each row in the table. All inserts and deletes to the table, and any updates to the key value affect at least one nonclustered index page (and adjoining pages in the page chain, if a page split or page deallocation takes place.) While locking a data page may slow access for a small number of rows, locks on frequently-used index pages can block access to a much larger set of rows. If a table averages 40 data rows per page, a lock on a data page restricts access to those 40 rows. If the leaf-level nonclustered index page that must be updated stores 100 keys and pointers, holding locks on that page restricts access to 4000 rows.

Avoiding "Hot Spots"

Hot spots occur when all updates take place on a certain page, as in a heap table, where all inserts happen on the last page of the page chain. For example, an unindexed history table that is updated by everyone will always have lock contention on the last page. This sample output from sp_sysmon shows that 11.9 percent of the inserts on a heap table need to wait for the lock:

Last Page	Locks on Heaps				
Granted		3.0	0.4	185	88.1 %
Waited		0.4	0.0	25	11.9 %

The best solution to this problem is to partition the history table. Partitioning a heap table creates multiple page chains in the table, and, therefore, multiple last pages for inserts. Concurrent inserts to

the table are less likely to block one another, since multiple last pages are available. Partitioning provides a way to improve concurrency for heap tables without creating separate tables for different groups of users. See “Improving Insert Performance with Partitions” on page 17-15 for information about partitioning tables.

Another solution for hot spots is to create a clustered index to distribute the updates across the data pages in the table. Like partitioning, this solution creates multiple insertion points for the table. However, it also introduces some overhead for maintaining the physical order of the table’s rows.

Decreasing the Number of Rows per Page

Another way to reduce contention is by decreasing the number of rows per page in your tables and indexes. When there is more empty space in the index and leaf 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 with max_rows_per_page

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.

The `max_rows_per_page` value 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, but the maximum cannot exceed 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

The `sp_chgattribute` system procedure 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 `bcp` utility as follows:
 - Copy out the table data.
 - Truncate the table.
 - Set the `max_rows_per_page` value with `sp_chgattribute`.
 - Copy the data back in.

Additional Locking Guidelines

These locking guidelines can help reduce lock contention and speed performance:

- Use the lowest level of locking required by each application, and use isolation level 3 only when necessary.

Updates by other transactions may be delayed until a transaction using isolation level 3 releases any of its shared locks at the end of the transaction. Use isolation level 3 only when nonrepeatable reads or phantoms may interfere with your desired results.

If only a few queries require level 3, use the `holdlock` keyword or `at isolation serializable` clause in those queries instead of using `set transaction isolation level 3` for the entire transaction. If most queries in the transaction require level 3, use `set transaction isolation level 3`, but use `noholdlock` or `at isolation read committed` in the remaining queries that can execute at isolation level 1.

- If you need to perform mass updates and deletes on active tables, you can reduce blocking by performing the operation inside a stored procedure using a cursor, with frequent commits.

- If your application needs to return a row, provide for user interaction, and then update the row, consider using timestamps and the `tsequal` function rather than `holdlock`.
- If you are using compliant third-party software, check the locking model in applications carefully for concurrency problems.

Also, other tuning efforts can help reduce lock contention. For example, if a process holds locks on a page, and must perform a physical I/O to read an additional page, it holds the lock much longer than it would have if the additional page had already been in cache. Better cache utilization or using large I/O can reduce lock contention in this case. Other tuning efforts that can pay off in reduced lock contention are improved indexing and good distribution of physical I/O across disks.

Reporting on Locks and Locking Behavior

The system procedures `sp_who`, `sp_lock`, and `sp_familylock` report on locks held by users and show processes that are blocked by other transactions.

Getting Information About Blocked Processes with `sp_who`

The system procedure `sp_who` reports on system processes. If a user's command is being blocked by locks held by another task or worker process, the `status` column shows "lock sleep" to indicate that this task or worker process is waiting for an existing lock to be released. The `blk` column shows the process ID of the entity holding the lock or locks.

If you do not provide a user name, `sp_who` reports on all processes in Adaptive Server. You can add a user name parameter to get `sp_who` information about a particular Adaptive Server user.

Viewing Locks with `sp_lock`

To get a report on the locks currently being held on Adaptive Server, use the system procedure `sp_lock`:

```
sp_lock
```

```
The class column will display the cursor name for locks associated  
with a cursor for the current user and the cursor id for other users.
```

fid	spid	locktype	table_id	page	dbname	class	context
0	7	Sh_intent	480004741	0	master	Non Cursor Lock	NULL
0	18	Ex_intent	16003088	0	sales	Non Cursor Lock	NULL
0	18	Ex_page	16003088	587	sales	Non Cursor Lock	NULL
0	18	Ex_page	16003088	590	sales	Non Cursor Lock	NULL
0	18	Ex_page	16003088	1114	sales	Non Cursor Lock	NULL
0	18	Ex_page	16003088	1140	sales	Non Cursor Lock	NULL
0	18	Ex_page	16003088	1283	sales	Non Cursor Lock	NULL
0	18	Ex_page	16003088	1362	sales	Non Cursor Lock	NULL
0	18	Ex_page	16003088	1398	sales	Non Cursor Lock	NULL
0	18	Ex_page-blk	16003088	634	sales	Non Cursor Lock	NULL
0	18	Update_page	16003088	1114	sales	Non Cursor Lock	NULL
0	18	Update_page-blk	16003088	634	sales	Non Cursor Lock	NULL
0	23	Sh_intent	16003088	0	sales	Non Cursor Lock	NULL
0	23	Sh_intent	176003658	0	sales	Non Cursor Lock	NULL
0	23	Ex_intent	208003772	0	sales	Non Cursor Lock	NULL
1	1	Sh_intent	176003658	0	stock	Non Cursor Lock	Sync-
pt duration request							
1	1	Sh_intent-blk	208003772	0	stock	Non Cursor Lock	Sync-
pt duration request							
1	8	Sh_page	176003658	41571	stock	Non Cursor Lock	NULL
1	9	Sh_page	176003658	41571	stock	Non Cursor Lock	NULL
1	10	Sh_page	176003658	41571	stock	Non Cursor Lock	NULL
11	11	Sh_intent	176003658	0	stock	Non Cursor Lock	Sync-
pt duration request							
11	12	Sh_page	176003658	41571	stock	Non Cursor Lock	NULL
11	13	Sh_page	176003658	41571	stock	Non Cursor Lock	NULL
11	14	Sh_page	176003658	41571	stock	Non Cursor Lock	NULL

This example shows the lock status of serial processes with *spids* 7, 18, and 23, and two families of processes: One family has a coordinating process with *spid* 1 and worker processes with *spids* 8, 9, and 10; the other family has a coordinating process with *spid* 11 and worker processes with *spids* 12, 13, and 14.

The *locktype* column indicates not only whether the lock is a shared lock (“Sh” prefix), an exclusive lock (“Ex” prefix), or an “Update” lock, but also whether it is held on a table (“table” or “intent”) or on a “page.”

A “blk” suffix indicates that this process is blocking another process that needs to acquire a lock. As soon as the blocking process completes, the other processes move forward. A “demand” suffix indicates that the process will acquire an exclusive lock as soon as all current shared locks are released.

In the *context* column, “Sync-pt duration request” means that the task is taking part in a parallel query, and that it will hold the lock until the query completes. The coordinating process always acquired a shared intent table lock that is held for the duration of the parallel

query. If the parallel query is part of a transaction, and earlier statements in the transaction performed data modifications, the coordinating process holds Sync-pt duration request locks on all of the changed data pages. Worker processes can hold Sync-pt duration request locks when the query operates at isolation level 3.

To see lock information about a particular login, give the *spid* for the process:

sp_lock 7

The class column will display the cursor name for locks associated with a cursor for the current user and the cursor id for other users.

fid	spid	locktype	table_id	page	dbname	class	context
0	7	Sh_intent	480004741	0	master	Non Cursor Lock	NULL

Viewing Locks with *sp_familylock*

The system procedure *sp_familylock* displays the locks held by a family. This examples shows that the coordinating process (*fid* 11, *spid* 11) holds a shared intent lock on the table and that each worker process holds a shared page lock:

sp_familylock 11

The class column will display the cursor name for locks associated with a cursor for the current user and the cursor id for other users.

fid	spid	locktype	table_id	page	dbname	class	context
11	11	Sh_intent	176003658	0	stock	Non Cursor Lock	Sync-pt duration request
11	12	Sh_page	176003658	41571	stock	Non Cursor Lock	NULL
11	13	Sh_page	176003658	41571	stock	Non Cursor Lock	NULL
11	14	Sh_page	176003658	41571	stock	Non Cursor Lock	NULL

Printing Deadlock Information to the Error Log

When deadlocks occur, Adaptive Server terminates one of the processes very quickly, making deadlocks difficult to observe with *sp_who* and *sp_lock*. You can use the configuration parameter *print deadlock information* to get more information. This parameter sends messages to the log and to the terminal session where the server was started. The messages in the log record the time and a sequential "Deadlock Id", like this:

```
00:00027:00018:1997/04/27 20:55:24.30 server  Deadlock Id 13 detected
```

In this output, *fid 27*, *spid 18* started the deadlock detection check, so its *fid* and *spid* values are used as the second and third values in the deadlock message.

The messages that are sent to the terminal window contain detailed information giving the family and process IDs, the commands, and tables involved in deadlocks. In the following report, *fid 27*, *spid 18* is deadlocked with *spid 51*. The deadlock involves pages in two tables, *customer* and *orders*, and *spid 51* is chosen as the deadlock victim:

```
00027:00018:Deadlock Id 13 detected. 1 deadlock chain(s) involved.

00027:00018:Process (Familyid 51,Spid 51) was executing a INSERT command
at line 1.
00027:00018:Process (Familyid 27,Spid 27) was executing a SELECT command
at line 1.
00027:00018:Process (Familyid 27, Spid 18) was waiting for a 'shared
page' lock on page 41273 of the 'customer' table in database 9 but
process (Familyid 51, Spid 51) already held a 'exclusive page' lock on
it.
00027:00018:Process (Familyid 51, Spid 51) was waiting for a
'exclusive intent' lock on the 'orders' table in database 9 but
process (Familyid 27, Spid 27) already held a 'shared intent' lock on
it.
00027:00018:Running owner Process (Familyid 27, Spid 27) belongs to
the same family as the deadlock initiator and holds a Sync-duration lock
that it wont release till the end of parallel execution.
00027:00018:
00027:00018:Process 51 was chosen as the victim. End of deadlock
information.
```

► **Note**

Setting **print deadlock information** to 1 can degrade Adaptive Server performance. For this reason, you should use it only when you are trying to determine the cause of deadlocks.

Observing Deadlocks Among Serial Queries

Deadlock checking locates deadlocks and resolves them by killing one of the processes in the time period configured by **deadlock checking period**, making deadlocks difficult to observe with system procedures. The **sp_who** and **sp_lock** output below was captured by setting **deadlock checking period** to perform deadlock checking approximately every 5 minutes. See “Delaying Deadlock Checking” on page 5-28 for more information.

◆ **WARNING!**

Do not experiment with deadlocks on a production server. Delaying deadlock checking can have a serious impact on performance.

In a deadlock that involves two serial tasks, `sp_who` shows “lock sleep” in the `status` column for both tasks. In the `blk` column, it shows that each is blocked by the other. The following `sp_who` output shows `spid 8` and `spid 11` have “lock sleep” status. The `blk` column indicates that `spid 8` is blocked by `spid 11` and that `spid 11` is blocked by `spid 8`.

```

sp_who
-----
fid spid status      loginame  origname  hostname  blk  dbname  cmd
-----
0    1 running      sa        sa        olympus   0    master  SELECT
0    2 sleeping    NULL     NULL     NULL     0    master  NETWORK HANDLER
0    3 sleeping    NULL     NULL     NULL     0    master  DEADLOCK TUNE
0    4 sleeping    NULL     NULL     NULL     0    master  MIRROR HANDLER
0    5 sleeping    NULL     NULL     NULL     0    master  HOUSEKEEPER
0    6 sleeping    NULL     NULL     NULL     0    master  CHECKPOINT SLEEP
0    7 recv sleep sa        sa        jove      0    master  AWAITING COMMAND
0    8 lock sleep delphi   delphi   olympus  11   stock  UPDATE
0    9 runnable  orion    orion    jasmine  0    stock  SELECT
0   10 running    pandora  pandora  olympus  23   stock  UPDATE
0   11 lock sleep apollo   apollo   jasmine  8    stock  UPDATE

```

`sp_lock` output gives you more information about the types of locks involved and the object on which they deadlocked. The `sp_lock` output below shows the `locktype` “`Ex_intent-blk`” for `spid 8` and `spid 11`. Together with the values in the `blk` column output, it shows that they are involved in a deadlock.

sp_lock

The `class` column will display the cursor name for locks associated with a cursor for the current user and the cursor id for other users.

```

sp_lock
-----
fid spid locktype      table_id  page  dbname  class          context
-----
0    7 Sh_page         16003088  0      master  Non Cursor Lock NULL
0    8 Ex_intent-blk  16003088  0      stock   Non Cursor Lock NULL
0    9 Sh_page         16003088  713    stock   Non Cursor Lock NULL
0   10 Ex_page         480004741 1662    stock   Non Cursor Lock NULL
0   11 Ex_intent-blk  480004741 0      stock   Non Cursor Lock NULL

```

Observing Deadlocks Among Worker Processes

Worker processes can acquire only shared locks, but it is still possible for them to be involved in deadlocks with processes that acquire

exclusive locks. The locks they hold meet one or more of these conditions:

- A coordinating process holds a table lock as part of a parallel query. Note that the coordinating process could hold exclusive locks on other tables as part of a previous query in a transaction.
- A parallel query is running at transaction isolation level 3 or using `holdlock` and holds locks.
- A parallel query is joining two or more tables while another process is performing a sequence of updates to the same tables within a transaction.

A single worker process can be involved in a deadlock such as the one shown earlier between two serial processes. For example, a worker process that is performing a join between two tables can deadlock with a serial process that is updating the same two tables. Deadlocks such as these print straightforward `sp_who` and `sp_lock` output, showing that they are blocking each other.

In some cases, deadlocks between serial processes and families involve a level of indirection that make them harder to detect in `sp_who` and `sp_lock` output. For example, if a task holds an exclusive lock on *tableA* and needs a lock on *tableB*, but a worker process holds a Sync point duration lock on *tableB*, the task must wait until the transaction that the worker process is involved in completes. If another worker process in the same family needs a lock on *tableA*, the result is a deadlock. Figure 5-12 illustrates the following deadlock scenario:

- The family identified by *fid* 8 is doing a parallel query that involves a join of *stock_tbl* and *sales*, at transaction level 3.
- The serial task identified by *spid* 17 (T₁₇) is performing inserts to *stock_tbl* and *sales* in a transaction.

These are the steps that lead to the deadlock:

- W_{8,9}, a worker process with a *fid* of 8 and a *spid* of 9, holds a shared lock on page 10862 of *stock_tbl*.
- T₁₇ holds an exclusive lock on page 634 of *sales_tbl*. T₁₇ needs an exclusive lock on page 10862, which it can not acquire until W_{8,9} releases its shared lock.
- The worker process W_{8,10} needs a shared lock on page 634, which it can not acquire until T₁₇ releases its exclusive lock.

The diagram illustrates that the result is a deadlock.

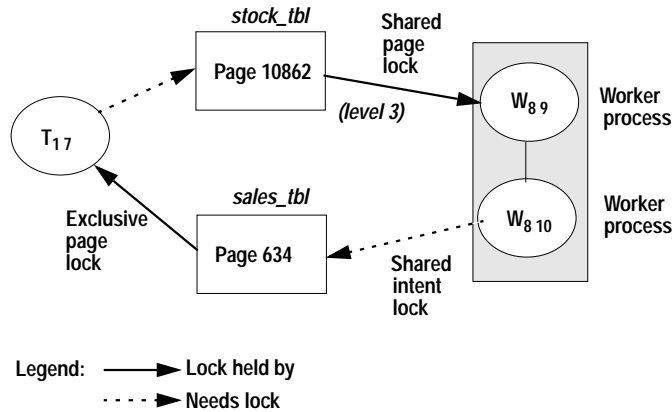


Figure 5-12: A deadlock involving a family of worker processes

sp_who output during such a deadlock shows a “sync sleep” status for at least one member of the family, typically the coordinating process.

The following sp_who output shows that spid 17 is blocked by the worker process with spid 9 and that spid 9 has a “sleeping” status, which means that this worker process (identified in the cmd column) will sleep, holding the lock, until fid 8 completes its transaction. Also notice that worker process W_{8 10} in the same family is blocked by spid 17.

```

sp_who
-----
fid  spid  status      loginame  origname  hostname  blk  dbname  cmd
-----
0    1  running    sa        sa        olympus   0    master  SELECT
0    2  sleeping   NULL     NULL     olympus   0    master  NETWORK HANDLER
0    3  sleeping   NULL     NULL     olympus   0    master  DEADLOCK TUNE
0    4  sleeping   NULL     NULL     olympus   0    master  MIRROR HANDLER
0    5  sleeping   NULL     NULL     olympus   0    master  HOUSEKEEPER
0    6  sleeping   NULL     NULL     olympus   0    master  CHECKPOINT SLEEP
0    7  recv sleep  sa        sa        jove      0    master  AWAITING COMMAND
0    17 lock sleep  apollo   apollo   jasmine   9    stock   UPDATE
8    8  sync sleep  delphi   delphi   olympus   0    stock   SELECT
8    9  sleeping  delphi   delphi   olympus   0    sales   WORKER PROCESS
8    10 lock sleep  delphi   delphi   olympus   17   stock   WORKER PROCESS
8    11 runnable delphi   delphi   olympus   0    stock   WORKER PROCESS
    
```

This indicates that it is possible for worker process W_{8 9} to be involved in a deadlock, even though there is no indication of that in the blk column for this task. To verify that there is a deadlock and to find out which processes are involved, you must look at sp_lock output.

`sp_lock` shows a value of “Sync-pt duration request” in the *context* column for worker processes `W8 10` and `W8 9`. The `sp_lock` output below shows that:

- `W8 9` holds a shared page lock request that is blocking another process.
- Serial process 17 holds an exclusive page lock that is blocking another process

Combined with the output of `sp_who`, this shows that worker processes `W8 10` and `W8 9` are involved in a deadlock with task 17.

`sp_lock`

The class column will display the cursor name for locks associated with a cursor for the current user and the cursor id for other users.

fid	spid	locktype	table_id	page	dbname	class	context
0	1	Sh_intent	480004741	0	master	Non Cursor Lock	NULL
8	8	Sh_intent	208003772	0	stock	Non Cursor Lock	Sync-pt
duration request							
8	9	Sh_page-blk	208003772	10862	stock	Non Cursor Lock	Sync-pt
duration request							
8	10	Sh_page	208003772	10862	stock	Non Cursor Lock	Sync-pt
duration request							
8	11	Sh_page	208003772	10862	stock	Non Cursor Lock	Sync-pt
duration request							
0	17	Ex_page-blk	16003088	634	sales	Non Cursor Lock	NULL

Observing Locks with `sp_sysmon`

Output from the system procedure `sp_sysmon` gives statistics on the page locks, table locks, and deadlocks discussed in this chapter.

Use the statistics to determine whether the Adaptive Server system is experiencing performance problems due to lock contention. For more information about `sp_sysmon` and lock statistics, see “Lock Management” on page 24-58.

Use Adaptive Server Monitor to pinpoint locking problems.

Intrafamily Blocking During Network Buffer Merges

When many worker processes are returning query results, you may see blocking between worker processes. This example shows five worker processes blocking on the sixth worker process:

```

sp_who 11
-----
fid  spid  status      loginame  origname  hostname  blk  dbname  cmd
-----
 11   11  sleeping    diana     diana     olympus   0    sales   SELECT
 11   16  lock sleep    diana     diana     olympus  18    sales   WORKER PROCESS
 11   17  lock sleep    diana     diana     olympus  18    sales   WORKER PROCESS
 11   18  send sleep    diana     diana     olympus   0    sales   WORKER PROCESS
 11   19  lock sleep    diana     diana     olympus  18    sales   WORKER PROCESS
 11   20  lock sleep    diana     diana     olympus  18    sales   WORKER PROCESS
 11   21  lock sleep    diana     diana     olympus  18    sales   WORKER PROCESS

```

Each worker process acquires an exclusive address lock on the network buffer while writing results to it. When the buffer is full, it is sent to the client, and the lock is held until the network write completes.

Configuring Locks and Lock Promotion Thresholds

A System Administrator can configure:

- The total number of locks available to processes on Adaptive Server.
- The lock promotion threshold, server-wide, for a database or for particular tables.
- The number of locks available per engine and the number of locks transferred between the global free lock list and the engines. See “freelock transfer block size” on page 11-53 and “max engine freelocks” on page 11-54 of the *System Administration Guide* for information on these parameters.

Configuring Adaptive Server's Lock Limit

Each lock counts toward Adaptive Server's limit of total number of locks. By default, Adaptive Server is configured with 5000 locks. A System Administrator can change this limit using the `sp_configure` system procedure. For example:

```
sp_configure "number of locks", 10000
```

You may also need to adjust the `sp_configure` parameter `total memory`, since each lock uses memory.

The number of locks required by a query can vary widely, depending on the number of concurrent and parallel processes and the types of actions performed by the transactions. Configuring the correct number for your system is a matter of experience and familiarity

with the system. You can start with an arbitrary number of 20 locks for each active concurrent connection, plus 20 locks for each worker process. Consider increasing the number of locks:

- If queries run at transaction isolation level 3, or use **holdlock**
- If you perform many multirow updates
- If you increase lock promotion thresholds

Setting the Lock Promotion Thresholds

The lock promotion thresholds set the number of page locks permitted by a task or worker process before Adaptive Server attempts to escalate to a table lock on the object. You can set the lock promotion threshold at the server-wide level, at the database level, and for individual tables. The default values provide good performance for a wide range of table sizes. Configuring the thresholds higher reduces the chance of queries acquiring table locks, especially for very large tables where queries lock hundreds of data pages.

Lock Promotion Occurs on a Per-Session Basis

Lock promotion occurs on a per-scan session basis. A **scan session** is the means by which Adaptive Server tracks table scans within a transaction. A single transaction can have more than one scan session for the following reasons:

- A table may be scanned more than once inside a single transaction in the case of joins, subqueries, **exists** clauses, and so on. Each scan of the table is a scan session.
- A query executed in parallel scans a table using multiple worker processes. Each worker process has a scan session.

A table lock is more efficient than multiple page locks when an entire table might eventually be needed. At first, Adaptive Server acquires page locks, then attempts to escalate to a table lock when a scan session acquires more page locks than the value set by the lock promotion threshold.

Since lock escalation occurs on a per-scan session basis, the total number of page locks for a single transaction can exceed the lock promotion threshold, as long as no single task or worker process acquires more than the lock promotion threshold number of page locks. Locks may persist throughout a transaction, so a transaction

that includes multiple scan sessions can accumulate a large number of locks.

Lock promotion from page locks to table locks cannot occur if a page lock owned by another Adaptive Server task or worker process conflicts with the type of table lock that is needed. For instance, if a task holds any exclusive page locks, no other process can promote to a table lock until the exclusive page locks are released. When this happens, a process can accumulate page locks in excess of the lock promotion threshold and exhaust all available locks in Adaptive Server. You may need to increase the value of the **number of locks** configuration parameter so that Adaptive Server does not run out of locks.

The three lock promotion parameters are **lock promotion HWM**, **lock promotion LWM**, and **lock promotion PCT**.

lock promotion HWM

lock promotion HWM (high water mark) sets a maximum number of locks allowed on a table before Adaptive Server escalates to a table lock. The default value is 200. When the number of locks acquired during a scan session exceeds this number, Adaptive Server attempts to acquire a table lock.

Setting **lock promotion HWM** to a value greater than 200 reduces the chance of any task or worker process acquiring a table lock on a particular table. For example, if a process updates more than 200 rows of a very large table during a transaction, setting the lock promotion high water mark higher keeps this process from attempting to acquire a table lock. Setting **lock promotion HWM** to less than 200 increases the chances of a particular task or worker process acquiring a table lock. Generally, it is good to avoid table locking, although it can be useful in situations where a particular user needs exclusive use of a table for which there is little or no contention.

lock promotion LWM

lock promotion LWM (low water mark) sets a minimum number of locks allowed on a table before Adaptive Server attempts to acquire a table lock. The default value is 200. Adaptive Server never attempts to acquire a table lock until the number of locks on a table is equal to **lock promotion LWM**. **lock promotion LWM** must be less than or equal to **lock promotion HWM**.

Setting lock promotion LWM parameter to a very high value decreases the chance for a particular task or worker process to acquire a table lock, which uses more page locks for the duration of the transaction, potentially exhausting all available locks in Adaptive Server. If this situation recurs, you may need to increase the value of the number of locks configuration parameter.

lock promotion PCT

lock promotion PCT sets the percentage of locked pages (based on the table size) above which Adaptive Server attempts to acquire a table lock when the number of locks is between the lock promotion HWM and the lock promotion LWM. The default value is 100.

Adaptive Server attempts to promote page locks to a table lock when the number of locks on the table exceeds:

$$(PCT * \text{number of pages in the table}) / 100$$

Setting lock promotion PCT to a very low value increases the chance of a particular user transaction acquiring a table lock. Figure 5-13 shows how Adaptive Server determines whether to promote page locks on a table to a table lock.

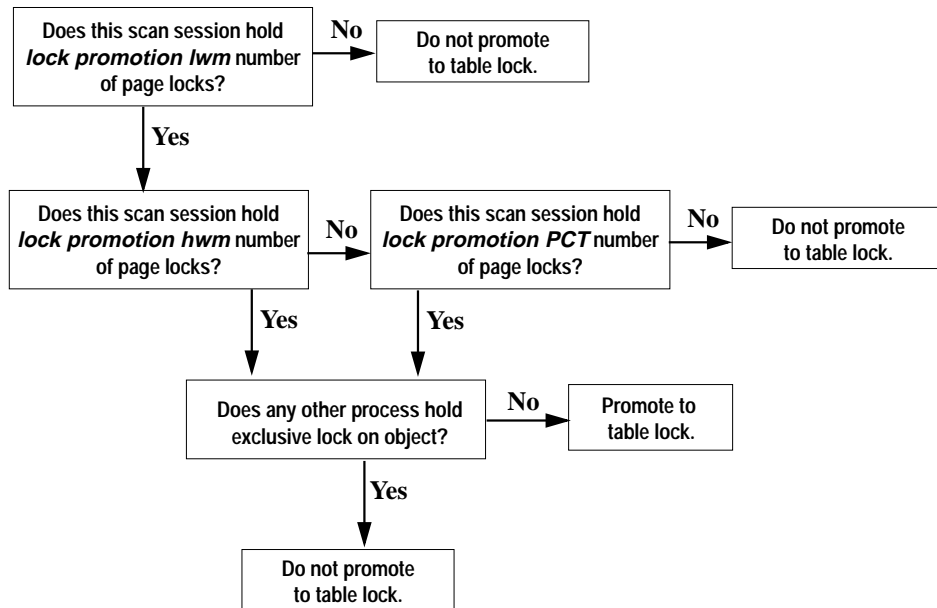


Figure 5-13: Lock promotion logic

Setting Lock Promotion Thresholds Server-Wide

The following command sets the server-wide lock promotion LWM to 100, the lock promotion HWM to 2000, and the lock promotion PCT to 50:

```
sp_setpglockpromote "server", null, 100, 2000, 50
```

In this example, Adaptive Server does not attempt to issue a table lock unless the number of locks on the table is between 100 and 2000. If a command requires more than 100 but less than 2000 locks, Adaptive Server compares the number of locks to the percentage of locks on the table. If the number of locks is greater than the number of pages resulting from the percentage calculation, Adaptive Server attempts to issue a table lock.

Adaptive Server calculates the number of pages as:

```
(PCT * number of pages in the table) / 100
```

The default value for lock promotion HWM (200) is likely to be appropriate for most applications. If you have many small tables with clustered indexes, where there is contention for data pages, you may be able to increase concurrency for those tables by tuning lock promotion HWM to 80 percent of number of locks.

The lock promotion thresholds are intended to maximize concurrency on heavily used tables. The default server-wide lock promotion threshold setting is 200.

Setting the Lock Promotion Threshold for a Table or Database

To configure lock promotion values for an individual table or database, initialize all three lock promotion thresholds. For example:

```
sp_setpglockpromote "table", titles, 100, 2000, 50
```

After the values are initialized, you can change any individual value. For example, to change the lock promotion PCT only, use the following command:

```
sp_setpglockpromote "table", titles, null, null, 70
```

To configure values for a database, use

```
sp_setpglockpromote "database", master, 1000,  
1100, 45
```

Precedence of Settings

You can change the lock promotion thresholds for any user database or an individual table. Settings for an individual table override the

database or server-wide settings; settings for a database override the server-wide values.

Server-wide values for lock promotion are represented by the `lock promotion HWM`, `lock promotion LWM`, and `lock promotion PCT` configuration parameters. Server-wide values apply to all user tables on the server, unless the database or tables have lock promotion values configured for them.

Dropping Database and Table Settings

To remove table or database lock promotion thresholds, use the `sp_droplockpromote` system procedure. When you drop a database's lock promotion thresholds, tables that do not have lock promotion thresholds configured will use the server-wide values. When you drop a table's lock promotion thresholds, Adaptive Server uses the database's lock promotion thresholds, if they have been configured, or the server-wide values, if the lock promotion thresholds have not been configured. You cannot drop the server-wide lock promotion thresholds.

Using `sp_sysmon` While Tuning Lock Promotion Thresholds

Use the system procedure `sp_sysmon` to see how many times lock promotions take place and the types of promotions they are. See "Lock Promotions" on page 24-64 for more information.

If there is a problem, look for signs of lock contention in the "Granted" and "Waited" data in the "Lock Detail" section of the `sp_sysmon` output. (See "Lock Detail" on page 24-60 for more information.) If lock contention is high and lock promotion is frequent, consider changing the lock promotion thresholds for the tables involved.

Use Adaptive Server Monitor to see how changes to the lock promotion threshold affect the system at the object level.

6

Determining or Estimating the Sizes of Tables and Indexes

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

This chapter contains the following sections:

- Tools for Determining the Sizes of Tables and Indexes 6-1
- Why Should You Care About the Sizes of Objects? 6-1
- Using `sp_spaceused` to Display Object Size 6-4
- Using `dbcc` to Display Object Sizes 6-6
- Using `sp_estspace` to Estimate Object Size 6-9
- Using Formulas to Estimate Object Size 6-11

Tools for Determining the Sizes of Tables and Indexes

Adaptive Server provides several tools that provide information on the current sizes of tables or indexes or that can predict future sizes:

- 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.
- The output of some `dbcc` commands reports on page usage as well as performing database consistency checks.

You can also compute the size using formulas provided in this chapter.

For partitioned tables, the system procedure `sp_helppartition` reports on the number of pages stored on each partition of the table. See Chapter 17, “Getting Information About Partitions,” for information.

Why Should You Care About the Sizes of Objects?

Knowing the sizes of your tables and indexes is key 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 7, “Indexing 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, using `dbcc traceon(302)` can often show why the optimizer made the decision.
- 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 17, “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.

System Table Sizes

When you create a database, the system tables are copied from the *model* database. At that time, the system tables contain very few rows and occupy very few data pages. Since each table and index is stored on a separate extent, this adds up to about 250 pages, or approximately 1/2MB. Most system tables remain fairly small—for example, *sysusers* for a database with 4000 users requires less than 70 pages. Some exceptions are:

- *syscomments* and *sysprocedures*—store the text and query plans for stored procedures and other compiled objects. If you use a large number of stored procedures and triggers, or very large and

complex procedures and triggers, these tables may become very large.

- *sysusermessages*—may become large if you create a large number of user-defined messages.
- *syslogs*—stores records for every transaction. It may become quite large, but it can be stored on a separate database device so that it does not contend with data.

Effects of Data Modifications on Object Sizes

The *sp_spaceused* and *dbcc* commands report actual space usage. The other methods presented in this chapter provide size estimates.

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 a table with a clustered index, and there is no room on the page for that row, the page splits, 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 and 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 percent full, unless the indexes are dropped and recreated periodically.

OAM Pages and Size Statistics

Information about the number of pages allocated to and used by an object is stored on the OAM pages for tables and indexes. This information is updated by most Adaptive Server processes when pages are allocated or deallocated. For a description of OAM pages, see “Object Allocation Map (OAM) Pages” on page 3-8.

The *sp_spaceused* system procedure reads these values to provide quick space estimates. Some *dbcc* commands update these values while they perform consistency checks.

`sp_spaceused` uses system functions to locate the size values on the OAM pages. Here is a simple query that uses the OAM page values, returning the number of used and reserved pages for all user tables (those with object IDs greater than 100):

```
select
  substring(object_name(id) + "." + name, 1,25) Name,
  data_pgs(id, doampg) "Data Used",
  reserved_pgs(id, doampg) "Data Res",
  data_pgs(id, ioampg) "Index Used",
  reserved_pgs(id, ioampg) "Index Res"
from sysindexes
where id > 100
```

Name	Data Used	Data Res	Index Used	Index Res
authors.authors	223	224	0	0
publishers.publishers	2	8	0	0
roysched.roysched	1	8	0	0
sales.sales	1	8	0	0
salesdetail.salesdetail	1	8	0	0
titleauthor.titleauthor	106	112	0	0
titles.title_id_cix	621	632	7	15
titles.title_ix	0	0	128	136
titles.type_price_ix	0	0	85	95
stores.stores	23	24	0	0
discounts.discounts	3	8	0	0
shipments.shipments	1	8	0	0
blurbs.blurbs	1	8	0	0
blurbs.tblurbs	0	0	7	8

The “Name” column reports the table and index name in the form “tablename.indexname”.

Using `sp_spaceused` to Display Object Size

The system procedure `sp_spaceused` reads values stored on the OAM page for an object 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 6-1 explains the headings in `sp_spaceused` output.

Table 6-1: 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 <i>data</i> , <i>index_size</i> , and <i>unused</i> .
<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.

If you want the sizes of the indexes to be reported separately, use this command:

```

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, 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, which are, by definition, the leaf level of a clustered index.

► **Note**

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 which is not currently used to store data.
- It provides detailed reports on the size of indexes and of *text* and *image* column storage.

Disadvantages of *sp_spaceused*

The disadvantages of *sp_spaceused* are:

- It may report inaccurate counts for row total and space usage.
- It does not correct inaccuracies, as *dbcc* does.
- It scans *sysindexes*, adding traffic to an already busy system table.
- Output is in kilobytes, while most query-tuning activity uses pages as a unit of measure.

Using *dbcc* to Display Object Sizes

Some of the *dbcc* commands that verify database consistency provide reports on space used by tables and indexes. Generally, these commands should not be used on production databases for routine space checking. They perform extensive I/O and some of them require locking of database objects. See Table 18-2 on page 18-17 of the *System Administration Guide* for a comparison of locking and performance effects.

If your System Administrator runs regular *dbcc* checks and saves results to a file, you can use this output to see the sizes of your objects and to track changes in table and index sizes.

If you want to use *dbcc* commands to report on table and index sizes, both the *tablealloc* and the *indexalloc* commands accept the *fast* option, which uses information in the OAM page, without performing

checks of all page chains. This reduces both disk I/O and the time that locks are held on your tables.

Table 6-2: dbcc commands that report space usage

Command	Arguments	Reports
dbcc tablealloc	Table name or table ID	Pages in a specified table and in each index on the table.
dbcc indexalloc	Table name or table ID and index ID	Pages in a specified index.
dbcc checkalloc	Database name, or current database, if no argument	Pages in all tables and indexes in the specified database. At the end of the report, prints a list of the allocation units in the database and the number of extents, used pages, and referenced pages in each allocation unit.

dbcc tablealloc(titles)

The default report option of OPTIMIZED is used for this run.

The default fix option of FIX is used for this run.

```

TABLE: titles                OBJID = 208003772
INDID=1  FIRST=2032          ROOT=2283          SORT=1
        Data level: 1.  864 Data Pages in 109 extents.
        Indid      : 1.  15 Index Pages in 3 extents.
INDID=2  FIRST=824          ROOT=827          SORT=1
        Indid      : 2.  47 Index Pages in 7 extents.
TOTAL # of extents = 119
Alloc page 2048 (# of extent=2 used pages=10 ref pages=10)
Alloc page 2304 (# of extent=1 used pages=7 ref pages=7)
Alloc page 1536 (# of extent=25 used pages=193 ref pages=193)
Alloc page 1792 (# of extent=27 used pages=216 ref pages=216)
Alloc page 2048 (# of extent=29 used pages=232 ref pages=232)
Alloc page 2304 (# of extent=28 used pages=224 ref pages=224)
Alloc page 256 (# of extent=1 used pages=1 ref pages=1)
Alloc page 768 (# of extent=6 used pages=47 ref pages=47)
Total (# of extent=119 used pages=930 ref pages=930) in this
database
DBCC execution completed. If DBCC printed error messages,
contact a user with System Administrator (SA) role.

```

The dbcc report shows output for *titles* with a clustered index (the information starting with "INDID=1") and a nonclustered index.

For the clustered index, **dbcc** reports both the amount of space used by the data pages themselves, 864 pages in 109 extents, and by the root and intermediate levels of the clustered index, 15 pages in 3 extents.

For the nonclustered index, **dbcc** reports the number of pages and extents used by the index.

Notice that some of the allocation pages are reported more than once in this output, since the output reports on three objects: the table, its clustered index, and its nonclustered index.

At the end of the output, **dbcc** reports the total number of extents used by the table and its indexes. The OAM pages and distribution pages are included.

You can use **dbcc indexalloc** to display the information for each index on the table. This example displays information about the nonclustered index on the *titles* table:

```
dbcc indexalloc(titles, 2)
```

```
The default report option of OPTIMIZED is used for this run.
The default fix option of FIX is used for this run.
*****
TABLE: titles          OBJID = 208003772
INDID=2  FIRST=824    ROOT=827      SORT=1
        Indid       : 2.  47 Index Pages in 7 extents.
TOTAL # of extents = 7
Alloc page 256 (# of extent=1 used pages=1 ref pages=1)
Alloc page 768 (# of extent=6 used pages=47 ref pages=47)
Total (# of extent=7 used pages=48 ref pages=48) in this database
DBCC execution completed. If DBCC printed error messages,
contact a user with System Administrator (SA) role.
```

The **dbcc checkalloc** command presents summary details for an entire database. Here is just the section that reports on the *titles* table:

```
TABLE: titles          OBJID = 208003772
INDID=1  FIRST=2032   ROOT=2283    SORT=1
        Data level: 1.  864 Data Pages in 109 extents.
        Indid       : 1.  15 Index Pages in 3 extents.
INDID=2  FIRST=824    ROOT=827      SORT=1
        Indid       : 2.  47 Index Pages in 7 extents.
TOTAL # of extents = 119
```


Advantages of *dbcc*

The advantages of using `dbcc` commands for checking the size of objects are:

- `dbcc` reports the amount of space used for each index and for the non-leaf portion of a clustered index.
- `dbcc` reports are in pages, which is convenient for most tuning work.
- `dbcc` reports the number of extents for each object, which is useful when estimating I/O using a 16K memory pool.
- `dbcc` reports are accurate. When `dbcc` completes, it updates the information on which `sp_spaceused` bases its reports.
- Using `dbcc tablealloc` or `indexalloc`, you can see how tables or indexes are spread across allocation units.
- You should run regular `dbcc` consistency checks of your databases. If you save the output to files, using this information to track space usage does not impact server performance.

Disadvantages of *dbcc*

The disadvantages of using `dbcc` commands for size checking are:

- `dbcc` can cause disk, data cache, and lock contention with other activities on the server.
- `dbcc` does not include space used by *text* or *image* columns.
- `dbcc` does not report on reserved pages, that is, pages in the extents that are allocated to the object, but which do not contain data. These pages cannot be used for other objects. It is possible to determine the number of reserved pages by multiplying the number of extents used by eight and comparing the result to the total number of used pages that `dbcc` reports.

Using *sp_estspace* to Estimate Object Size

`sp_spaceused` and `dbcc` commands report on actual space usage. The system procedure `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. It 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. You provide an estimate of the number of rows.

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` has additional features that allow you to specify a `fillfactor`, the average size of variable-length fields and text fields, and the I/O speed. For more information, see `sp_estspace` 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 plan for table and index growth.
- `sp_estspace` provides a page count at each index level and helps you estimate the number of index levels.
- `sp_estspace` estimates the amount of time needed to create the index.
- `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 basic 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. Due to the 32 bytes

required for the page header, 2016 bytes are available for data on a 2K page.

► **Note**

Do not confuse this figure with the maximum row size, which is 1960 bytes, due to overhead in other places in Adaptive Server.

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 `fillfactor` or `max_rows_per_page` in a `create index` statement changes the final size of an index. See “Effects of Setting `fillfactor` to 100 Percent” on page 6-25, and “`max_rows_per_page` Value” on page 6-28.

The formulas in this section use the maximum size for variable-length character and binary data. If you want to use the average size instead of the maximum size, see “Using Average Sizes for Variable Fields” on page 6-26.

If your table includes `text` or `image` datatypes, use 16 (the size of the text pointer that is stored in the row) in your calculations. Then see “text and image Data Pages” on page 6-28 to see how to calculate the storage space required for the actual `text` or `image` data.

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 “page utilization percent” on page 11-38 in the *System Administration Guide*.

Storage Sizes for Datatypes

The storage sizes for Adaptive Server datatypes are shown in Table 6-3:

Table 6-3: Storage sizes for Adaptive Server datatypes

Datatype	Size
<i>char</i>	Defined size
<i>nchar</i>	Defined size * @@ <i>ncharsize</i>
<i>varchar</i>	Actual number of characters
<i>nvarchar</i>	Actual number of characters * @@ <i>ncharsize</i>
<i>binary</i>	Defined size
<i>varbinary</i>	Data size
<i>int</i>	4
<i>smallint</i>	2
<i>tinyint</i>	1
<i>float</i>	4 or 8, depending on precision
<i>double precision</i>	8
<i>real</i>	4
<i>numeric</i>	2–17, depending on precision and scale
<i>decimal</i>	2–17, depending on precision and scale
<i>money</i>	8
<i>smallmoney</i>	4
<i>datetime</i>	8
<i>smalldatetime</i>	4
<i>bit</i>	1
<i>text</i>	16 bytes + 2K * number of pages used
<i>image</i>	16 bytes + 2K * number of pages used
<i>timestamp</i>	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*, *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 6-26.

Calculating the Sizes of Tables and Clustered Indexes

The formulas and examples are divided into two sections:

- Steps 1–6 outline the calculations for a table with a clustered index, giving the table size and the size of the index tree. The example that follows Step 6 illustrates the computations on a table that has 9,000,000 rows.
- Steps 7–12 outline the calculations for computing the space required by nonclustered indexes, followed by another example on the 9,000,000-row table.

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.

Use the first formula if all the columns are fixed length, and defined as NOT NULL. Use the second formula if the row contains variable-length columns or columns defined as NULL.

Fixed-Length Columns Only

Use this formula if the table contains only fixed-length columns:

$$\begin{array}{r}
 4 \quad (\text{Overhead}) \\
 + \quad \text{Sum of bytes in all fixed-length columns} \\
 \hline
 = \text{Data row size}
 \end{array}$$

Some Variable-Length Columns

Use this formula if the table contains variable-length columns or columns that allow null values:

$$\begin{array}{r}
 4 \quad \text{(Overhead)} \\
 + \quad \text{Sum of bytes in all fixed-length columns} \\
 + \quad \text{Sum of bytes in all variable-length columns} \\
 \hline
 = \text{Subtotal} \\
 \\
 + \quad (\text{Subtotal} / 256) + 1 \quad \text{(Overhead)} \\
 + \quad \text{Number of variable-length columns} + 1 \\
 + \quad 2 \quad \text{(Overhead)} \\
 \hline
 = \text{Data row size}
 \end{array}$$

Step 2: Compute the Number of Data Pages

$$2016 / \text{Data row size} = \text{Number of data rows per page}$$

$$\text{Number of rows} / \text{Rows per page} = \text{Number of data pages required}$$

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

$$\begin{array}{r}
 5 \quad (\text{Overhead}) \\
 + \quad \text{Sum of bytes in the fixed-length index keys} \\
 \hline
 = \text{Clustered row size}
 \end{array}$$

Some Variable-Length Columns

$$\begin{array}{r}
 5 \quad (\text{Overhead}) \\
 + \quad \text{Sum of bytes in the fixed-length index keys} \\
 + \quad \text{Sum of bytes in variable-length index keys} \\
 \hline
 = \text{Subtotal}
 \end{array}$$

$$\begin{array}{r}
 + \quad (\text{Subtotal} / 256) + 1 \quad (\text{Overhead}) \\
 + \quad \text{Number of variable-length columns} + 1 \\
 + \quad 2 \quad (\text{Overhead}) \\
 \hline
 = \text{Clustered index row size}
 \end{array}$$

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

Step 4: Compute the Number of Clustered Index Pages

$$(2016 / \text{Clustered row size}) - 2 = \text{No. of clustered index rows per page}$$

$$\text{No. of rows} / \text{No. of CI rows per page} = \text{No. of index pages at next level}$$

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:

$$\begin{array}{r}
 \text{No. of index pages} \\
 \text{at last level}
 \end{array}
 /
 \begin{array}{r}
 \text{No. of clustered} \\
 \text{index rows per page}
 \end{array}
 =
 \begin{array}{r}
 \text{No. of index pages at} \\
 \text{next level}
 \end{array}$$

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:

Index Levels	Pages
2	
1	+
0	+
	<hr/>
	Total number of index pages

Step 6: 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:

$$\begin{aligned} \text{Number of reserved data pages} / 63,750 &= \text{Minimum OAM pages} \\ \text{Number of reserved data pages} / 2000 &= \text{Maximum OAM pages} \end{aligned}$$

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

$$\begin{aligned} \text{Number of reserved index pages} / 63,750 &= \text{Minimum OAM pages} \\ \text{Number of reserved index pages} / 2000 &= \text{Maximum OAM pages} \end{aligned}$$

Total Pages Needed

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

	Minimum	Maximum
Clustered index pages		
OAM pages	+	+
Data pages	+	+
OAM pages	+	+
Total		

Example: Calculating the Size of a 9,000,000-Row Table

The following example computes the size of the data and clustered index for a table containing:

- 9,000,000 rows
- Sum of fixed-length columns = 100 bytes
- Sum of 2 variable-length columns = 50 bytes
- Clustered index key, fixed length, 4 bytes

Calculating the Data Row Size (Step 1)

The table contains variable-length columns.

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

Calculating the Number of Data Pages (Step 2)

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

$$2016 / 160 = 12 \text{ Data rows per page}$$

$$9,000,000 / 12 = 750,000 \text{ Data pages}$$

Calculating the Clustered Index Row Size (Step 3)

	5	(Overhead)		
+	4	Sum of bytes in the fixed-length index keys		
	9			= Clustered Row Size

Calculating the Number of Clustered Index Pages (Step 4)

$$(2016 / 9) - 2 = 222 \text{ Clustered Index Rows Per Page}$$

$$750,000 / 222 = 3379 \text{ Index Pages (Level 0)}$$

$$3379 / 222 = 16 \text{ Index Pages (Level 1)}$$

$$16 / 222 = 1 \text{ Index Page (Level 2)}$$

Calculating the Total Number of Index Pages (Step 5)

Index Levels	Pages	Rows
2	1	16
1	+	16
0	+	3379
Index total:	3396	750000
Data total:	750000	9000000

Calculating the Number of OAM Pages and Total Pages (Step 6)

Both the table and the clustered index require one or more OAM pages.

For Data Pages:

$$750,000 / 63,750 = 12 \text{ (minimum)}$$

$$750,000 / 2000 = 376 \text{ (maximum)}$$

For Index Pages:

$$3379 / 63,750 = 1 \text{ (minimum)}$$

$$3379 / 2000 = 2 \text{ (maximum)}$$

Total Pages Needed:

	Minimum	Maximum
Clustered index pages	3379	3379
OAM pages	1	2
Data pages	750000	750000
OAM pages	12	376
Total	753392	753757

Calculating the Size of Nonclustered Indexes**Step 7: Calculate the Size of the Leaf Index Row**

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

Use this formula if the index contains only fixed-length keys:

$$\begin{array}{r}
 7 \text{ (Overhead)} \\
 + \\
 \hline
 \text{Sum of fixed-length keys} \\
 = \text{Size of leaf index row}
 \end{array}$$

Some Variable-Length Keys

Use this formula if the index contains any variable-length keys:

$$\begin{array}{r}
 9 \quad \text{(Overhead)} \\
 + \quad \text{Sum of length of fixed-length keys} \\
 + \quad \text{Sum of length of variable-length keys} \\
 + \quad \text{Number of variable-length keys} + 1 \\
 \hline
 = \text{Subtotal} \\
 \\
 + \quad (\text{Subtotal} / 256) + 1 \text{ (overhead)} \\
 \hline
 = \text{Size of leaf index row}
 \end{array}$$

Step 8: Calculate the Number of Leaf Pages in the Index

$$2016 / \text{Size of leaf index row} = \text{No. of leaf rows per page}$$

$$\text{No. of rows in table} / \text{No. of leaf rows per page} = \text{No. of leaf pages}$$

Step 9: Calculate the Size of the Non-Leaf Rows

$$\begin{array}{r}
 \text{Size of leaf index row} \\
 + \quad 4 \quad \text{Overhead} \\
 \hline
 = \text{Size of non-leaf row}
 \end{array}$$

Step 10: Calculate the Number of Non-Leaf Pages

$$(\text{2016} / \text{Size of non-leaf row}) - 2 = \text{No. of non-leaf index rows per page}$$

$$\text{No. of leaf pages at last level} / \text{No. of non-leaf index rows per page} = \text{No. of index pages at next level}$$

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:

$$\begin{array}{r} \text{No. of index pages} \\ \text{at last level} \end{array} / \begin{array}{r} \text{No. of non-leaf index} \\ \text{rows per page} \end{array} = \begin{array}{r} \text{No. of index pages at} \\ \text{next level} \end{array}$$

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
4	
3	+
2	+
1	+
0	+

	Total number of 2K data pages used

Step 12: Calculate Allocation Overhead and Total Pages

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

Number of index pages / 2000 = Maximum OAM pages

Total Pages Needed

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

	Minimum	Maximum
Nonclustered index pages		
OAM pages	+	+
Total	_____	

Example: Calculating the Size of a Nonclustered Index

The following example computes the size of a nonclustered index on the 9,000,000-row table used in the preceding example. There are two keys, one fixed length and one variable length.

- 9,000,000 rows
- Sum of fixed-length columns = 100 bytes
- Sum of 2 variable-length columns = 50 bytes
- Composite nonclustered index key:
 - Fixed length column, 4 bytes
 - Variable length column, 20 bytes

Calculate the Size of the Leaf Index Row (Step 7)

The index contains variable-length keys:

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

Calculate the Number of Leaf Pages (Step 8)

$2016 / 36 = 56$ Nonclustered leaf rows per page

$9,000,000 / 56 = 160,715$ leaf pages

Calculate the Size of the Non-Leaf Rows (Step 9)

$$\begin{array}{r}
 36 \text{ Size of leaf index row (from Step 7)} \\
 + \quad 4 \text{ Overhead} \\
 \hline
 40 = \text{Size of non-leaf index rows}
 \end{array}$$

Calculate the Number of Non-Leaf Pages (Step 10)

$$\begin{array}{ll}
 (2016 / 40) - 2 = 48 & \text{Non-leaf index rows per page} \\
 160715 / 48 = 3349 & \text{Index pages, level 1} \\
 3348 / 48 = 70 & \text{Index pages, level 2} \\
 70 / 48 = 2 & \text{Index pages, level 3} \\
 2 / 48 = 1 & \text{Index page, level 4}
 \end{array}$$

Calculating Totals (Step 11)

Index Levels	Pages	Rows
4	1	2
3	2	70
2	70	3348
1	3349	160715
0	160715	9000000
	<hr/>	
	164137	

Calculating OAM Pages Needed (Step 12)

$$164137 / 63,750 = 3 \text{ (minimum)}$$

$$164137 / 2000 = 83 \text{ (maximum)}$$

Total Pages Needed

	Minimum	Maximum
Index pages	164137	164137
OAM pages	3	83
Total pages	164140	164220

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 **fillfactor** value used when an index is created
- Whether computations used average row size or maximum row size
- Very small text rows
- **max_rows_per_page** value
- Use of *text* and *image* data

Effects of Setting *fillfactor* to 100 Percent

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 (Step 4) and when calculating the number of non-leaf pages (Step 9). Both of these calculations subtract 2 from the number of rows per page. Eliminate the -2 from these calculations.

► **Note**

The **fillfactor** value affects the size of an index when it is created. Fillfactors are not maintained as tables are updated. Use **fillfactor** adjustments for read-only tables.

Other *fillfactor* Values

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 6-15 and “Step 8: Calculate the Number of Leaf Pages in the Index” on page 6-21.

Distribution Pages

Distribution pages are created when you create an index on existing data and when you run `update statistics`. A distribution page occupies one full data page. Distribution pages are essential for proper functioning of the optimizer.

Using Average Sizes for Variable Fields

All of the formulas use the maximum size of the variable-length fields.

One way to determine the average size of the fields in an existing table is:

```
select avg(datalength(column_name))
       from table_name
```

This query performs a table scan, which may not be acceptable in a production system or for a very large table. You can limit the number of rows it searches by placing the column value in a temporary table, with `set rowcount`, and running the query on the temporary table.

You can use the average value in Steps 1 and 4 in calculating table size, and in Step 6 in calculating the nonclustered index size. You will need slightly different formulas:

In Step 1

Use the sum of the **average** length of the variable-length columns instead of the sum of the defined length of the variable-length columns to determine the average data row size. “Step 1: Calculate the Data Row Size” on page 6-14.

In Step 2

Use the average data row size in the first formula. See “Step 2: Compute the Number of Data Pages” on page 6-15.

In Step 3

You must perform the addition twice. The first time, calculate the maximum index row size, using the given formula. The second time, calculate the average index row size, substituting the sum of the **average** number of bytes in the variable-length index keys for the sum of the defined number of bytes in the variable-length index keys. See “Step 3: Compute the Size of Clustered Index Rows” on page 6-15.

In Step 4

Substitute this formula for the first formula in Step 4, using the two length values:

$$(2016 - 2 * \text{maximum_length}) / \text{average_length} = \text{No. of clustered index rows per page}$$

See “Step 4: Compute the Number of Clustered Index Pages” on page 6-16.

In Step 6

You must perform the addition twice. The first time, calculate the maximum leaf index row size, using the given formula. The second time, calculate the average leaf index row size, substituting the **average** number of bytes in the variable-length index keys for the **sum** of byte in the variable-length index keys. See “Step 6: Calculate Allocation Overhead and Total Pages” on page 6-17.

In Step 7

Use the average leaf index row size in the first division procedure. See “Step 7: Calculate the Size of the Leaf Index Row” on page 6-20.

In Step 8

Use the average leaf index row size. See “Step 8: Calculate the Number of Leaf Pages in the Index” on page 6-21

In Step 9

Substitute this formula for the first formula in Step 9, using the maximum and averages calculated in Step 6:

$$(2016 - 2 * \text{Maximum_length}) / \text{Average_length} = \text{No. of non-leaf index rows per page}$$

See “Step 9: Calculate the Size of the Non-Leaf Rows” on page 6-21.

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 will be:

$$\text{Number of Rows} / 256 = \text{Number of data pages required}$$

***max_rows_per_page* Value**

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 6-15 and “Step 8: Calculate the Number of Leaf Pages in the Index” on page 6-21.

***text* and *image* Data Pages**

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

text and *image* columns are designed to 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 *text* or an *image* column is defined to allow null values, and the row is created with an *insert* statement that includes NULL for the *text* or *image* column, the column is not initialized, and the storage is not allocated.

If a *text* or an *image* 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 *text* or *image* column is subsequently set to NULL, a single page remains allocated.

Each text or image page stores 1800 bytes of data. To calculate the number of text chain 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.

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.

Tuning Query Performance

7

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.

This chapter contains the following sections:

- How Indexes Can Affect Performance 7-1
- Symptoms of Poor Indexing 7-2
- Index Limits and Requirements 7-6
- Tools for Query Analysis and Tuning 7-6
- Indexes and I/O Statistics 7-9
- Estimating I/O 7-16
- Optimizing Indexes for Queries That Perform Sorts 7-23
- Choosing Indexes 7-28
- Techniques for Choosing Indexes 7-36
- Index Statistics 7-39
- How the Optimizer Uses the Statistics 7-42
- Index Maintenance 7-43
- Displaying Information About Indexes 7-46
- Tips and Tricks for Indexes 7-46
- Choosing Fillfactors for Indexes 7-48

How Indexes Can 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. In general, if there is not a good reason to have an index, it should not be there.

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 table access completely, when an index covers a query
- Use ordered data to avoid sorts

In addition, you can create indexes to enforce the uniqueness of data and to randomize the storage location of inserts.

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. Since Adaptive Server cannot know whether one row or several rows match the search arguments, it cannot stop when it finds a matching row, but must read every row in the table.

If a query is searching for a unique value in a table that has 600 data pages, this requires 600 disk reads. If an index points to the data value, the query could 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. Even if this response time is acceptable for the query in question, heavy disk I/O by one query has a negative impact on overall throughput.

Table scans may occur:

- When there is no index on the search arguments for a query
- When there is an index, but the optimizer determines that it is not useful
- When there are no search arguments

Detecting Indexing Problems

Lack of indexing, or incorrect indexing, results in poor query performance. Some of the major indications are as follows:

- 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.

The underlying problems relating to indexing and poor performance are as follows:

- No indexes are assigned to a table, so table scans are always used to retrieve data.
- An existing index is not selective enough for a particular query, so it is not used by the optimizer.
- The index does not support a critical range query, so table scans are required.
- Too many indexes are assigned to a table, so data modifications are slow.
- The index key is too large, so using the index generates high I/O.

These underlying problems are described in the following sections.

Lack of Indexes Is Causing Table Scans

If select operations and joins take too long, it is likely that an appropriate index does not exist or is not being used by the optimizer. Analyzing the query can help you determine whether another index is needed, whether an existing index can be modified, or whether the query can be modified to use an existing index. If an index exists, but is not being used, careful analysis of the query and the values used is required to determine the source of the problem.

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 help optimize range queries. Range queries that reference the keys of noncovering nonclustered indexes use the index for ranges that return a limited number of rows. As the range and the number of rows the query returns increase, however, using a nonclustered index to return the rows can cost more than a table scan.

As a rule, if access via a nonclustered index returns more rows than there are pages in the table, the index is more costly than the table scan (using 2K I/O on the table pages). If the table scan can use 16K I/O, the optimizer is likely to choose the table scan when the number of rows to be returned exceeds the number of I/Os (number of pages divided by 8).

► **Note**

The optimizer estimates the cost of both physical and logical I/O for the query, as well as other costs. The optimizer does not use the calculations above to estimate query costs.

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. Each time an insert, update, or delete operation affects an index key, the leaf level, (and sometimes higher levels) of a nonclustered index need to be updated. Analyze the requirements for each index and try to eliminate those that are unnecessary or rarely used.

Index Entries Are Too Large

Large index entries cause large indexes, so try to keep them 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. Nonmatching index scans can be very expensive.

The following example uses `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 (c1 char(10),
                      c2 char(20),
                      c4 char(40))

create index t1 on demotable(c1)
create index t2 on demotable(c2)
create index t4 on demotable(c4)

sp_estspace demotable, 500000
```

name	type	idx_level	Pages	Kbytes
demotable	data	0	15204	37040
t1	nonclustered	0	4311	8623
t1	nonclustered	1	47	94
t1	nonclustered	2	1	2
t2	nonclustered	0	6946	13891
t2	nonclustered	1	111	222
t2	nonclustered	2	3	6
t2	nonclustered	3	1	2
t4	nonclustered	0	12501	25002
t4	nonclustered	1	339	678
t4	nonclustered	2	10	20
t4	nonclustered	3	1	2

Total_Mbytes

83.58

name	type	total_pages	time_mins
t1	nonclustered	4359	25
t2	nonclustered	7061	34
t4	nonclustered	12851	53

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. A nonmatching index scan on the leaf level of *t2* would require 6946 disk reads, compared to 4311 for the index on *t1*.

► **Note**

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

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 stored in order by index key.
- You can create a maximum of 249 nonclustered indexes per table.
- A key can be made up of multiple columns. The maximum is 16 columns. The maximum number of bytes per index key is 600.
- When you create a clustered index, Adaptive Server requires an additional 120 percent of the table size in the database. It must create a copy of the table and allocate space for the root and intermediate pages for the index. Note that 120 percent is a general rule; if you have very long keys, you may need even more space.
- 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.

Tools for Query Analysis and Tuning

The query analysis tools that you use most often while tuning queries and indexes are listed in Table 7-1.

Table 7-1: Tools for managing index performance

Tool	Function
<code>set showplan on</code>	Shows the query plan for a query, including indexes selected, join order, and worktables. See Chapter 9, “Understanding Query Plans.”
<code>set statistics io on</code>	Shows how many logical and physical reads and writes are performed to process the query. If you have enabled resource limits, it also shows total actual I/O cost. See “Indexes and I/O Statistics” on page 7-9.

Table 7-1: Tools for managing index performance (continued)

Tool	Function
<code>set statistics time on</code>	Shows how long it takes to execute the query.
<code>set noexec on</code>	Usually used with <code>set showplan on</code> , this command suppresses execution of the query. You see the plan the optimizer would choose, but the query is not executed. <code>noexec</code> is useful when the query would return very long results or could cause performance problems on a production system. Note that output from <code>statistics io</code> is not shown when <code>noexec</code> is in effect (since the query does not perform I/O).
<code>dbcc traceon (302)</code>	This special trace flag lets you see the calculations used by the optimizer to determine whether indexes should be used. See “Tuning with <code>dbcc traceon 302</code> ” on page 10-17.

Tools that provide information on indexes or help in tuning indexes are listed in Table 7-2.

Table 7-2: Additional tools for managing index performance

Tool	Function
<code>sp_configure "default fill factor percent"</code>	Sets or displays the default fillfactor for index pages.
<code>sp_help, sp_helpindex</code>	Provides information on indexes that exist for a table.
<code>sp_estspace</code>	Provides estimates of table and index size, the number of pages at each level of an index, and the time needed to create each index.
<code>sp_spaceused</code>	Provides information about the size of tables and its indexes.
<code>update statistics</code>	Updates the statistics kept about distribution and density of keys in an index.

The commands that provide information on space usage are described in Chapter 6, “Determining or Estimating the Sizes of Tables and Indexes.”

Table 7-3 lists additional commands.

Table 7-3: Advanced tools for query tuning

Tool	Function
set forceplan	Forces the query to use the tables in the order specified in the from clause.
set table count	Increases the number of tables that the optimizer considers at one time while determining join order.
select, delete, update clauses: (index...prefetch...mru_lru)	Specifies the index, I/O size, or cache strategy to use for the query.
set prefetch	Toggles prefetch for query tuning experimentation.
sp_cachestrategy	Sets status bits to enable or disable prefetch and fetch-and-discard cache strategies.
(parallel <i>degree</i>)	Specifies the degree of parallelism for a query.

The tools listed in Table 7-3 are described in Chapter 10, “Advanced Optimizing Techniques.”

Figure 7-1 shows how many of these tools are related to the process of running queries in Adaptive Server.

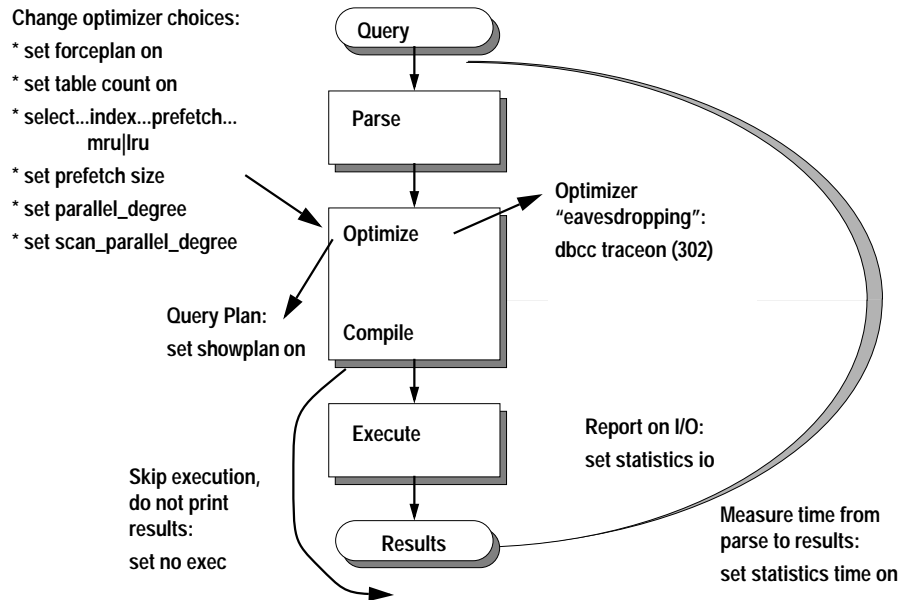


Figure 7-1: Query processing analysis tools and query processing

Using *sp_sysmon* to Observe the Effects of Index Tuning

Use the system procedure *sp_sysmon* (or Adaptive Server Monitor) as you work on index tuning. Look at the output for improved cache hit ratios, a reduction in the number of physical reads, and fewer context switches for physical reads. For more information about using *sp_sysmon*, see Chapter 24, "Monitoring Performance with *sp_sysmon*."

Indexes and I/O Statistics

The *statistics io* option of the *set* command reports information about physical and logical I/O and the number of times a table was accessed. Reports from *set statistics io* follow the query results and provide actual I/O performed by the query.

For each table in a query, including worktables, *statistics io* reports one line of information with several values for the pages read by the query and one row that reports the total number of writes. If a

System Administrator has enabled resource limits, `statistics io` also includes a line that reports the total actual I/O cost for the query. The following example shows `statistics io` output for a query with resource limits enabled:

```
select avg(total_sales)
from titles
```

```
Table: titles  scan count 1,  logical reads: (regular=656 apf=0
total=656), physical reads: (regular=444 apf=212 total=656),
apf IOs used=212
Total actual I/O cost for this command: 13120.
Total writes for this command: 0
```

The following sections describe the four major components of `statistics io` output:

- Actual I/O cost
- Total writes
- Read statistics
- Table name and “scan count”

Total Actual I/O Cost Value

If resource limits are enabled, `statistics io` prints the “Total actual I/O cost” line. Adaptive Server reports the total actual I/O as a unitless number. The formula for determining the cost of a query is:

$$\text{IO cost} = 2 * \text{logical IOs} + 18 * \text{physical IOs}$$

This formula multiplies the cost of a logical I/O by the number of logical I/Os and the cost of a physical I/O by the number of physical I/Os. For more information on these numbers, see “How Is “Fast” Determined?” on page 8-4.

Statistics for Writes

`statistics io` reports the total number of buffers written by the command. Read-only queries may report writes when they cause pages to move past the wash marker in the cache so that the write on the dirty page starts. Queries that change data may report only a single write, the log page write, because the changed pages remain in the MRU section of the data cache.

Statistics for Reads

Figure 7-4 shows the values that `statistics io` reports for logical and physical reads.

Table 7-4: `statistics io` output for reads

Output	Description
logical reads	
regular	Number of times that a page needed by the query was found in cache (holding the cache spinlock)
apf	Number of times that an asynchronous prefetch (APF) request was found to be already in cache. Only the number of times that the search required holding the cache spinlock are counted; if the page is found in cache without holding the spinlock, it is not counted.
total	Sum of regular and apf logical reads
physical reads	
regular	Number of times a buffer was brought into cache by regular asynchronous I/O
apf	Number of times that a buffer was brought into cache by APF
total	Sum of regular and apf physical reads
apf IOs used	Number of buffers brought in by APF in which one or more pages were used during the query.

Sample Output With and Without an Index

Using `statistics io` to perform a query on a table without an index and the same query on the same table with an index shows how important good indexes can be to query and system performance. Here is a sample query:

```
select title
from titles
where title_id = "T5652"
```

statistics io Without an Index

With no index on `title_id`, `statistics io` reports these values, using 2K I/O:

```
Table: titles scan count 1, logical reads:(regular=624 apf=0 total=624),
physical reads:(regular=230 apf=394 total=624), apf IOs used=394
Total actual I/O cost for this command: 12480.
Total writes for this command: 0
```

This output shows that:

- The query performed a total of 624 logical I/Os, all regular logical I/Os.
- The query performed 624 physical reads. Of these, 230 were regular asynchronous reads, and 394 were asynchronous prefetch reads.

statistics io With an Index

With a clustered index on *title_id*, *statistics io* reports these values for the same query, also using 2K I/O:

```
Table: titles scan count 1, logical reads: (regular=3 apf=0 total=3),
physical reads: (regular=3 apf=0 total=3), apf IOs used=0
Total actual I/O cost for this command: 60.
Total writes for this command: 0
```

The output shows that:

- The query performed 3 logical reads
- The query performed 3 physical reads: 2 reads for the index pages and 1 read for the data page

Adding the index improves performance by a factor of 200.

Scan Count

The line that reports reads also reports the scan count for the query. The scan count shows the number of times a table or index was used in the query. A “scan” can represent any of these access methods:

- A table scan.
- An access via a clustered index. Each time the query starts at the root page of the index and follows pointers to the data pages, it is counted.
- An access via a nonclustered index. Each time the query starts at the root page of the index and follows pointers to the leaf level of the index (for a covered query) or to the data pages, it is counted.
- If queries run in parallel, each worker process is counted as a scan.

You need to use `showplan`, as described in Chapter 9, “Understanding Query Plans,” to determine which access method is used.

Queries Reporting a Scan Count of 1

Examples of queries that return a scan count of 1 are:

- A point query:

```
select title_id
from titles
  where title_id = "T55522"
```

- A range query:

```
select au_lname, au_fname
from authors
  where au_lname > "Smith"
  and au_lname < "Smythe"
```

If the columns in the `where` clauses of these queries are indexed, the queries can use the indexes to probe the tables; otherwise, they perform table scans. In either case, they require only a single probe of the table to return the required rows.

Queries Reporting a Scan Count of More Than 1

Examples of queries that return larger scan count values are:

- Parallel queries report a scan count for each worker process.
- Queries that have indexed `where` clauses connected by `or` report a scan for each `or` clause. This query has an index on `title_id` and another on `pub_id`:

```
select title_id
from titles
  where title_id = "T55522"
  or pub_id = "P302"
```

It reports a scan count of 2:

```
Table: titles scan count 2, logical reads: (regular=6 apf=0
total=6), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
```

Note that if any `or` clause is not indexed, the query performs a single table scan.

- In joins, inner tables are scanned once for each qualifying row in the outer table. In the following example, the outer table, `publishers`, has three `publishers` with the state “NY”, so the inner table, `titles`, reports a scan count of 3:

```

select title_id
from titles t, publishers p
where t.pub_id = p.pub_id
and p.state = "NY"

```

Table: titles scan count 3, logical reads:(regular=1869 apf=0 total=1869), physical reads: (regular=286 apf=337 total=623), apf IOs used=337

Table: publishers scan count 1, logical reads:(regular=2 apf=0 total=2), physical reads: (regular=2 apf=0 total=2), apf IOs used=0

This query performs table scans on both tables. *publishers* occupies only 2 data pages, so 2 physical I/Os are reported. There are 3 matching rows in *publishers*, so the query scans *titles* 3 times, reporting 1869 logical reads (623 pages * 3).

Queries Reporting Scan Count of 0

Multistep queries and certain other types of queries may report a scan count of 0. Some examples are:

- Those that perform deferred updates
- select...into queries
- Some queries that create worktables

Deferred Updates and Scan Count = 0

Deferred updates perform the changes to the data in two steps:

- Finding the rows using appropriate indexes. This step has a scan count of 1 or more. Log records are written to the transaction log during this step.
- Changing the data pages. This step is labeled "scan count 0".

If there is no index on *title_id*, this query is done in deferred mode:

```

update titles set title_id = "T47166"
where title_id = "T33040"

```

Table: titles scan count 0, logical reads: (regular=0 apf=0 total=0), physical reads: (regular=0 apf=0 total=0), apf IOs used=0

Table: titles scan count 1, logical reads:(regular=623 apf=0 total=623), physical reads: (regular=286 apf=337 total=623), apf IOs used=337

Total writes for this command: 1

This command, which inserts data that is selected from the same table, is also performed in deferred mode and reports a scan count of 0:

```

insert pubtab
select * from pub_table

```

Table: pubtab scan count 0, logical reads:(regular=129 apf=0 total=129),
 physical reads: (regular=1 apf=0 total=1), apf IOs used=0
 Table: pubtab scan count 1, logical reads: (regular=16 apf=0 total=16),
 physical reads: (regular=8 apf=9 total=17), apf IOs used=8
 Total writes for this command: 20

Scan Count for insert...select and select...into Commands

The insert...select and select...into commands that work in direct mode report I/O for scan count 0 on the target table:

```

select *
into pub_table
from publishers

```

Table: publishers scan count 1, logical reads:(regular=2 apf=0 total=2),
 physical reads: (regular=2 apf=0 total=2), apf IOs used=0
 Table: newpub scan count 0, logical reads: (regular=31 apf=0 total=31),
 physical reads: (regular=0 apf=0 total=0), apf IOs used=0
 Total writes for this command: 7

Worktables and a Scan Count of 0

Queries that include order by and distinct sometimes create worktables and sort the results. The I/O on these worktables is reported with a scan count equal to 0:

```

select distinct state
from authors

```

Table: authors scan count 1, logical reads:(regular=224 apf=0 total=224),
 physical reads: (regular=8 apf=216 total=224), apf IOs used=216
 Table: Worktable1 scan count 0, logical reads: (regular=5099 apf=0 total=5099),
 physical reads: (regular=0 apf=0 total=0), apf IOs used=0
 Total writes for this command: 0

Relationship Between Physical and Logical Reads

If a page needs to be read from disk, it is counted as a physical read and a logical read. Logical I/O is always greater than or equal to physical I/O.

Logical I/O always reports 2K data pages. Physical reads and writes are reported in buffer-sized units. Multiple pages that are read in a single I/O operation are treated as a unit: They are read, written, and moved through the cache as a single buffer.

Logical Reads, Physical Reads, and 2K I/O

With 2K I/O, the number of times that a page is found in cache for a query is logical reads minus physical reads. When the total number of logical reads and physical reads is the same for a table scan, it means that each page was read from disk and accessed only once during the query.

Often, when indexes are used to access a table, or when you are rerunning queries during testing, `statistics io` reports a different value for the total number of logical and physical reads.

Physical Reads and Large I/O

Physical reads are not reported in pages, but in buffers, that is, the actual number of times Adaptive Server accesses the disk. If the query uses 16K I/O (as reported by `showplan`), a single physical read brings 8 data pages into cache. If a query reports 100 16K physical reads, it has read 800 data pages. If the query needs to scan each of those data pages, it reports 800 logical reads. If a query, such as a join query, must read the page multiple times because other I/O has flushed the page from the cache, each physical read is counted.

Reads and Writes on Worktables

Reads and writes are reported for any worktable that needs to be created for the query. When a query creates more than one worktable, the worktables are numbered in `statistics io` output to correspond to the worktable numbers used in `showplan` output.

Effects of Caching on Reads

If you are testing a query and checking its I/O, and you execute the same query a second time, you may get surprising physical read values, especially if the query uses LRU replacement strategy. The first execution reports a high number of physical reads; the second execution reports 0 physical reads.

The first time you execute the query, all the data pages are read into cache and remain there until other server processes flush them from the cache. Depending on the cache strategy used for the query, the pages may remain in cache for a longer or shorter period of time.

- If the query uses the fetch-and-discard (MRU) cache strategy, the pages are read into the cache at the wash marker. In small or very

active caches, pages read into the cache at the wash marker are flushed fairly quickly.

- If the query uses LRU cache strategy to read the pages in at the top of the MRU end of the page chain, the pages remain in cache for longer periods of time. This is especially likely to happen if you have a large data cache and the activity on your server is low.

For more information on testing and cache performance, see “Testing Data Cache Performance” on page 16-9.

Estimating I/O

Checking the output from set statistics provides information when you actually execute a query. However, if you know the approximate sizes of your tables and indexes, you can make I/O estimates without running queries. Once you know the sizes of your tables and indexes, and the number of index levels in each index, you can quickly determine whether I/O performance for a query is reasonable or whether a particular query needs tuning efforts.

Following are some guidelines and formulas for making I/O estimates.

Table Scans

When a query requires a table scan, Adaptive Server:

- Reads each page of the table from disk into the data cache
- Checks the data values (if there is a `where` clause) and returns matching rows

If the table is larger than your data cache, Adaptive Server keeps flushing pages out of cache, so that it can read in additional pages, until it processes the entire query. It may use one of two strategies:

- **Fetch-and-discard (MRU) replacement strategy:** If the optimizer estimates that the pages will not be needed again for a query, it reads the page into the cache just before the wash marker. Pages remain in cache for a short time and do not flush other, more heavily used pages out of cache, since the more frequently used pages are repeatedly placed at the MRU end of the cache.
- **LRU replacement strategy:** Pages replace a “least recently used” buffer and are placed on the most recently used end of the chain. They remain in cache until other disk I/O flushes them from the cache.

Table scans are performed:

- When no index exists on the columns used in the query.
- When the optimizer chooses not to use an index. It makes this choice when it determines that using the index is more expensive than performing a table scan. This is more likely with nonclustered indexes. The optimizer may determine that it is faster to read the table pages directly than it is to go through several levels of indexes for each row that is to be returned.

As a general rule, table scans are chosen over nonclustered index access when the query returns more rows than there are pages in the table, when using 2K I/O. For larger I/O sizes, the calculation is number of pages returned divided by number of pages per I/O.

Evaluating the Cost of a Table Scan

Performance of a table scan depends on:

- Table size, in pages
- Speed of I/O
- Data cache sizes and bindings
- I/O size available for the cache

The larger the cache available to the table, the more likely it is that all or some of the table pages will be in memory because of a previous read. But the optimizer makes the pessimistic assumption: It cannot know what pages are in cache, so it assumes that the query must perform all the physical I/O required.

To estimate the cost of a table scan:

1. Determine the number of pages that need to be read. Use `sp_spaceused`, `dbcc tablealloc`, or `sp_estspace` to check the number of pages in the table, or use `set statistics io on` and execute a query that scans the table using 2K I/O.
2. Determine the I/O size available in your data cache. Execute `sp_help tablename` to see if the table is bound to a cache and `sp_cacheconfig` to see the I/O sizes available for that cache.

Divide the number of pages in the table by the number of pages that can be read in one I/O.

Determine the number of disk I/Os that your system can perform per second; divide the total reads by that number, as shown in the formula in Figure 7-2.

$$\text{I/O time} = \frac{\text{Pages in table/pages per IO}}{\text{Disk reads per second}}$$

Figure 7-2: Formula for computing table scan time

For example, if `sp_estspace` gives a table size of 76,923 pages, and your system reads 50 pages per second into 2K buffers, the time required to execute a table scan on the table is:

$$76923 \text{ pages} / 50 \text{ reads per second} = 1538 \text{ seconds, or about 25 minutes}$$

If your cache can use 16K buffers, the value is:

$$\begin{aligned} 76,923 \text{ pages} / 8 \text{ pages per read} &= 9615 \text{ reads} \\ 9615 \text{ reads} / 50 \text{ reads per second} &= 192 \text{ seconds, or about 3 minutes} \end{aligned}$$

The actual speed would be better if some of the data were in cache.

Evaluating the Cost of Index Access

If you are selecting a specific value, the index can be used to go directly to the row containing that value, making fewer comparisons than it takes to scan the entire table. In range queries, the index can point to the beginning and the end of a range. This is particularly true if the data is ordered by a clustered index or if the query uses a covering nonclustered index.

When Adaptive Server estimates that the number of index and data page I/Os is less than the number required to scan the entire table, it uses the index.

To determine the number of index levels, use one of these methods:

- Use `sp_estspace`, giving the current number of rows in the table, or perform space estimates using the formulas.
- Use `set statistics io` and run a point query that returns a single row using the clustered index. The number of levels is the number of logical reads minus 1.

Evaluating the Cost of a Point Query

A point query that uses an index performs one I/O for each index level plus one read for the data page. In a frequently used table, the root page and intermediate pages of indexes are often found in cache. In that case, physical I/O is reduced by one or two reads.

Evaluating the Cost of a Range Query

Range queries may be optimized very differently, depending on the type of index. Range queries on clustered indexes and on covering nonclustered indexes are very efficient. They use the index to find the first row and then scan forward on the leaf level.

Range queries using nonclustered indexes (those that do not cover the query) are more expensive, since the rows may be scattered across many data pages. The optimizer always estimates that access via a nonclustered index requires a physical I/O for each row that needs to be returned.

Range Queries Using Clustered Indexes

To estimate the number of page reads required for a range query that uses the clustered index to resolve a range query, you can use the formula shown in Figure 7-3.

$$\text{Reads required} = \text{Number of index levels} + \frac{\text{\# of rows returned/\# of rows per page}}{\text{Pages per IO}}$$

Figure 7-3: Computing reads for a clustered index range query

If a query returns 150 rows, and the table has 10 rows per page, the query needs to read 15 data pages, plus the needed index pages. If the query uses 2K I/O, it requires 15 or 16 I/Os for the data pages, depending on whether the range starts in the middle of a page. If your query uses 16K I/O, these 15 data pages require a minimum of 2 or 3 I/Os for the database. 16K I/O reads entire extents in a single I/O, so 15 pages might occupy 2 or 3 extents if the page chains are contiguous in the extents. If the page chains are not contiguous, because the table has been frequently updated, the query could require as many as 15 or 16 16K I/Os to read the entire range. See “Maintaining Data Cache Performance for Large I/O” on page 16-33 for more information on large I/O and fragmentation.

Figure 7-4 shows how a range query using a clustered index uses the index to find the first matching row on the data pages. The next-page pointers are used to scan forward until a nonmatching row is encountered.

```
select fname, lname, id
from employees
where lname between "Greaves"
and "Highland"
Clustered index on lname
```

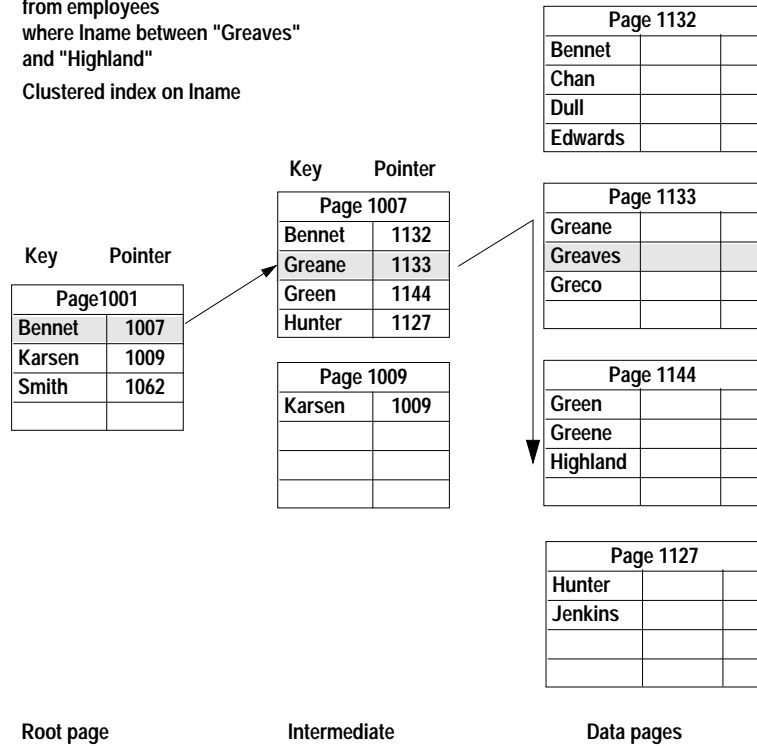


Figure 7-4: Range query on a clustered index

Range Queries with Covering Nonclustered Indexes

Range queries using covering nonclustered indexes can perform very well:

- The index can be used to position the search at the first qualifying row in the index.
- Each index page contains more rows than corresponding data rows, so fewer pages need to be read.

- Index pages tend to remain in cache longer than data pages, so fewer physical I/Os are needed.
- If the cache used by the nonclustered index allows large I/O, up to 8 pages can be read per I/O.
- The data pages do not have to be accessed.

Figure 7-5 shows a range query on a covering nonclustered index.

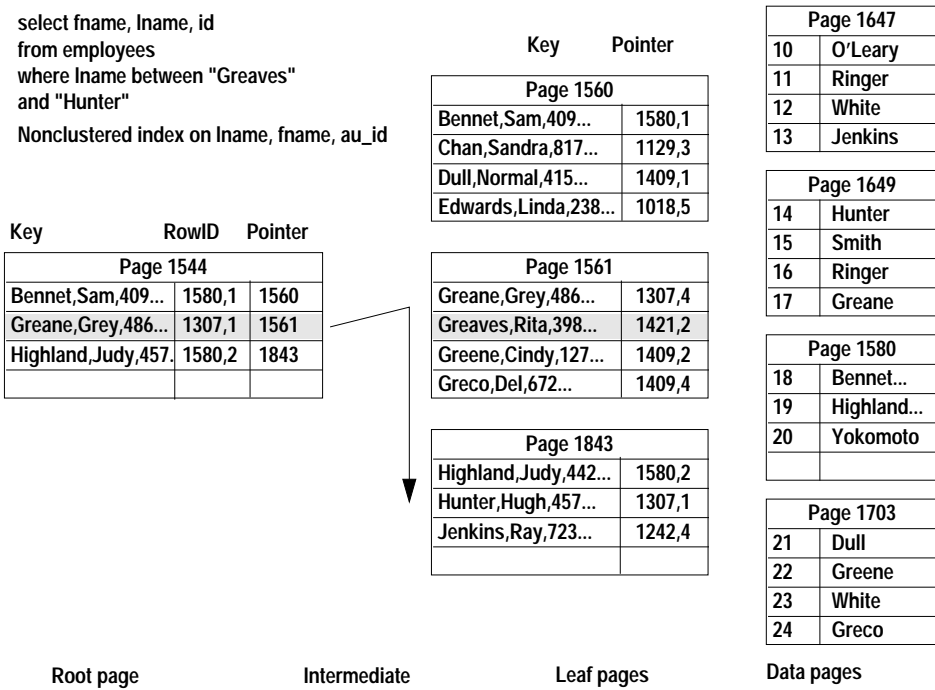


Figure 7-5: Range query with a covering nonclustered index

To estimate the cost of using a covering nonclustered index, you need to know:

- The number of index levels
- The number of rows per page on the leaf level of the index
- The number of rows that the query returns

- The number of leaf pages read per I/O, as calculated using the formula in Figure 7-6.

$$\text{Reads required} = \text{Number of index levels} + \frac{\text{\# of rows returned/\# of rows per page}}{\text{Pages per IO}}$$

Figure 7-6: Computing reads for a covering nonclustered index range query

Range Queries with Noncovering Nonclustered Indexes

Clustered indexes and covering nonclustered indexes generally perform extremely well for range queries on the leading index key, because they scan leaf pages that are in order by index key. However, range queries on the key of noncovering nonclustered indexes are much more sensitive to the size of the range in the query. For small ranges, a nonclustered index may be efficient, but for larger ranges, using a nonclustered index can require more reads than a table scan.

At the leaf level of a nonclustered index, the keys are stored sequentially, but at the data level, rows can be randomly placed throughout the data pages. The keys on a single leaf page in a nonclustered index can point to a large number of data rows. When Adaptive Server returns rows from a table using a nonclustered index, it performs these steps:

- It locates the first qualifying row at the leaf level of the nonclustered index.
- It follows the pointers to the data page for that index.
- It finds the next row on the index page, and locates its data page. The page may already be in cache, or it may have to be read from disk.

When you run this query on the *authors* table in the *pubtune* database, with an index on *au_lname*, it selects 265 rows, performing a table scan of the table's 223 data pages:

```
select au_fname, au_lname, au_id
from authors
where au_lname between "Greaves"
and "Highland"
```

Especially with short keys, a single leaf-level page in a nonclustered index can point to 100 or more data pages. Using a clustered index, Adaptive Server follows the index pointers to the first data page with the correct value, and then follows the page chain to read subsequent

rows. However, with a nonclustered index, the data pages for a given range can be scattered throughout the database, so it may be necessary to read each page several times. The formula for estimating I/O for range queries accessing the data through a nonclustered index is shown in Figure 7-7.

$$\text{Reads required} = \text{Number of index levels} + \frac{\text{\# of rows returned}}{\text{\# of rows per leaf level index page}} + \text{\# of rows returned}$$

Figure 7-7: Computing reads for a nonclustered index range query

The optimizer estimates that a range query that returns 500 rows, with an index structure of 3 levels and 100 rows per page on the leaf level of the nonclustered index, requires 507 or 508 I/Os:

- 1 read for the root level and 1 read for the intermediate level
- 5 or 6 reads for the leaf level of the index
- 500 reads for the data pages

Although it is possible that some of the rows in the result set will be found on the same data pages, or that they will be found on data pages already in cache, this is not predictable. The optimizer costs a physical I/O for each row to be returned, and if this estimate exceeds the cost of a table scan, it chooses the table scan. If the table in this example has less than 508 pages, the optimizer chooses a table scan.

Optimizing Indexes for Queries That Perform Sorts

Queries that perform sorts are executed in different ways, depending on the availability of indexes. For example, the optimizer chooses one of these access methods for a query with an `order by` clause:

- When there are no indexes on the table, Adaptive Server performs a table scan to find the correct rows, copies the rows into a worktable, and sorts the results.
- When there is no index on the column(s) in the `order by` clause, but there are useful indexes for other clauses, Adaptive Server uses the cheapest index to locate the rows, copies the rows to a worktable, and sorts the results.
- When there is an index that includes the column(s) in the `order by` clause, Adaptive Server determines whether using that index can help avoid the cost of copying rows to a worktable and sorting them. Very often, using the index that matches the `order by` clause is cheaper, but in some cases, the optimizer chooses to create and sort a worktable because the index that matches the `order by` clause is more expensive than using another index and performing the sort.

► **Note**

Parallel sort operations are optimized very differently for partitioned tables. See Chapter 15, “Parallel Sorting,” for more information.

How the Optimizer Costs Sort Operations

When Adaptive Server optimizes queries that require sorts:

- It computes the cost of using an index that matches the required sort order, if such an index exists.
- It computes the physical and logical I/O cost of creating a worktable and performing the sort for every index where the index order does not match the sort order.
- It computes the physical and logical I/O cost of performing a table scan, creating a worktable, and performing the sort.

Adding the cost of creating and sorting the worktable to the cost of index access and the cost of creating and sorting the worktable favors the use of an index that supports the `order by` clause.

For composite indexes, the order of the keys in the index must match the order of the columns named in the order by clause.

Sorts and Clustered Indexes

If the data is clustered in the order required by the sort, the sort is not needed and is not performed. If the entire table must be scanned, Adaptive Server follows the index pointers for the first row in the table to the first page and scans forward, following the page pointers. If a where clause on the index key limits the search, the index is used to position the search, and then the scan continues on the leaf pages, as shown in Figure 7-8.

```
select fname, lname, id
from employees
where lname between "Dull"
and "Greene"
order by lname
Clustered index on lname
```

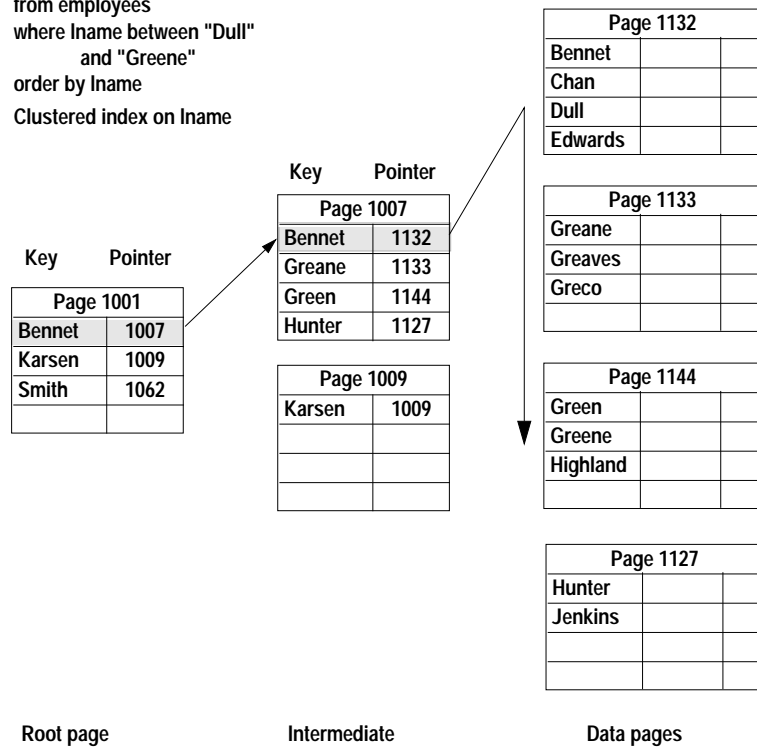


Figure 7-8: An order by query using a clustered index

The following range query on the *titles* table returns about 2000 rows. It can use a clustered index on *title_id* to position the search at the first

matching row, thereby reducing physical and logical I/O, and avoiding the sort step:

```
select * from titles
where title_id between 'T43' and 'T791'
order by title_id
```

Table: titles scan count 1, logical reads:(regular=258 apf=0 total=258), physical reads: (regular=8 apf=336 total=344), apf IOs used=249 Total writes for this command: 0

Queries requiring descending sort order (for example, order by title_id desc) can avoid sorting by scanning pages in reverse order. If the entire table is needed for a query without a where clause, Adaptive Server follows the index pointers to the last page, and then scans backward using the previous page pointers. If the where clause includes an index key, the index is used to position the search, and then the pages are scanned backward, as shown in Figure 7-9.

```
select fname, lname, id
from employees
where lname <= "Highland"
order by lname desc
Clustered index on lname
```

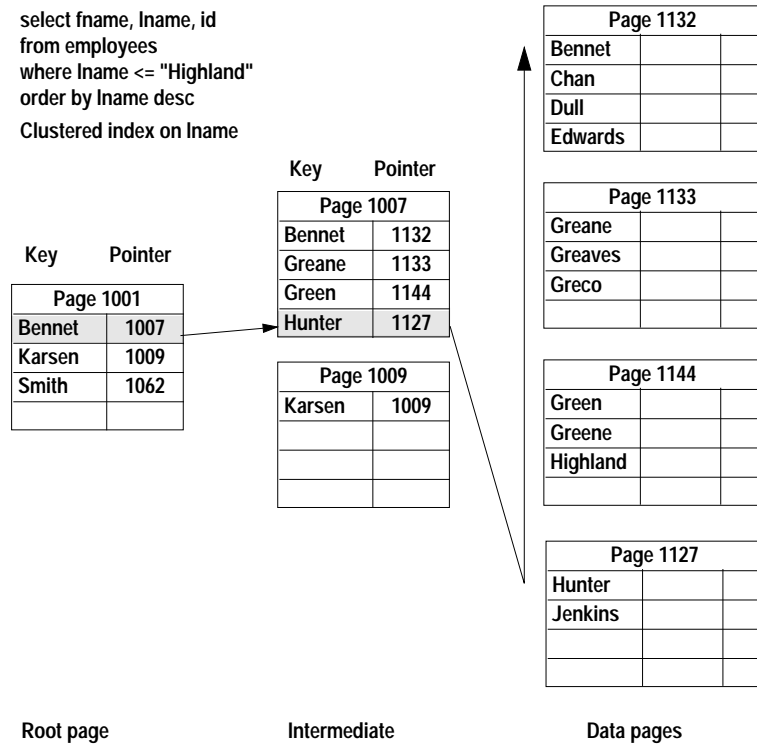


Figure 7-9: An order by desc query using a clustered index

For more information on descending scans see “Descending Scans and Sorts” on page 7-27.

Sorts and Nonclustered Indexes

With a nonclustered index, Adaptive Server determines whether using the nonclustered index is faster than performing a table scan, inserting rows into a worktable, and then sorting the data.

To use an index that matches the sort order, Adaptive Server needs to retrieve the index pages and use them to access the proper data pages in the sort order. If the number of rows to be returned is small, nonclustered index access is cheaper than a table scan.

If *where* clauses on other columns match other indexes on the table, Adaptive Server costs the effects of using those indexes and performing the sort. If the index that matches the sort order is not very selective, but another index that matches another *where* clause would return very few rows, the total cost of the query would still be lower if the sort were performed. For example, if a *where* clause returned only a few rows, but did not match the sort order, it would still be cheaper to use that index and sort the rows than to use an index that accesses hundreds of rows, but does not require a sort.

The optimizer also computes the cost of performing a table scan, constructing a worktable, and performing the sort. If many rows are being returned, the optimizer may choose to perform the table scan and sort the worktable, especially if large I/O is available for performing the table scan.

The following returns 2000 rows. With a nonclustered index on *title_id*, it uses a worktable to sort the results:

```
select * from titles
where title_id between 'T43' and 'T791'
order by title_id
```

The query performs a table scan, reading all 626 pages of the table, and produces this report from *statistics io*:

```
Table: titles scan count 3, logical reads: (regular=2098 apf=0
total=2098), physical reads: (regular=333 apf=293 total=626), apf IOs
used=292
Table: Worktable1 scan count 0, logical reads: (regular=2235 apf=0
total=2235), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0
```

This is the same query that requires only 344 physical reads and no worktable and sort when it uses a clustered index to return the 2000

rows—but for a nonclustered index, the I/O estimate would be 2000 physical I/Os to use the index and avoid the sort. For a smaller range of values, however, the nonclustered index could be used effectively.

Sorts When the Index Covers the Query

When all the columns named in the select list, the search arguments, and the **group by** and **order by** clause are included in a nonclustered index, Adaptive Server can use the leaf level of the nonclustered index to retrieve the data without reading the data pages.

If the **order by** columns form a **prefix subset** of the index keys, the rows are returned directly from the nonclustered index leaf pages. If the columns do not form a prefix subset of the index keys, a worktable is created and sorted.

With a nonclustered index on *au_lname*, *au_fname*, *au_id* of the *authors* table, this query can return the data directly from the leaf pages:

```
select au_id, au_lname
from authors
order by au_lname, au_fname
```

Table: authors scan count 1, logical reads: (regular=91 apf=0 total=91), physical reads: (regular=8 apf=83 total=91), apf IOs used=83
Total writes for this command: 0

This query performs only 91 physical I/Os, instead of the 448 physical I/Os that would be required to scan the entire base table.

Descending Scans and Sorts

The optimizer can avoid sorts for queries that use **desc** in an **order by** clause to return results in descending order. To execute these queries, Adaptive Server scans the page chain in reverse order. If the index has a composite key, **desc** must be specified for each key named in the **order by** clause to use this optimization. For example, this query can use a clustered index on *au_lname*, *au_fname* to avoid a sort:

```
select au_lname, au_fname
from authors
order by au_lname desc, au_fname desc
```

Descending Scans and Joins

If two or more tables are being joined, and the **order by** clause specifies descending order for index keys on the joined tables, any of the

tables involved can be scanned in descending order to avoid the worktable and sort costs. If all the columns for one table are in ascending order, and the columns for the other tables are in descending order, the first table is scanned in ascending order and the others in descending order.

When Sorts Are Still Needed

Sorts are also required for result sets when the columns in the result set are a superset of the index keys. For example, if the index on *authors* includes *au_lname* and *au_fname*, and the *order by* clause also includes the *au_id*, the query would require a sort.

Deadlocks and Descending Scans

Descending scans may deadlock with queries performing update operations using ascending scans and with queries performing page splits and shrinks, except when the backward scans are performed at transaction isolation level 0.

The configuration parameter *allow backward scans* controls whether the optimizer uses the backward scan strategy. The default value of 1 allows descending scans. See “allow backward scans” on page 11-92 of the *System Administration Guide* for more information on this parameter. Also, see “Index Scans” on page 24-55 for information on the number of ascending and descending scans performed and “Deadlocks by Lock Type” on page 24-62 for information on detecting deadlocks.

Choosing Indexes

Questions to ask when working with index selection are:

- What indexes are associated currently with a given table?
- What are the most important processes that make use of the table?
- What is the overall ratio of select operations to data modifications performed on the table?
- Has a clustered index been assigned to 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 the indexes support the joins and referential integrity checks?
- Does indexing affect update types (direct vs. 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 vs. select operations
- Performance requirements of reports vs. 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.

Guidelines for Clustered Indexes

These are general guidelines for clustered indexes:

- Most 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, the clustered index key should not be placed on a monotonically 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.
- Clustered indexes provide very good performance when the key matches the search argument in range queries, such as:

```
where colvalue >= 5 and colvalue < 10
```
- Other good choices for clustered index keys are columns used in `order by` and `group by` clauses and in joins.
- If possible, do not include frequently updated columns as keys in clustered indexes. 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

► **Note**

If the primary key is a monotonically increasing value, placing a clustered index on this key can cause contention for the data page where the inserts take place. This severely limits concurrency. Be sure that your clustered index key randomizes the location of inserts.

- Columns that are accessed by range, such as:

```
col1 between "X" and "Y" or col12 > "X" and < "Y"
```
- Columns used by `order by` or `group 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,”

places where data modification activity encounters blocking due to locks on data or index pages.

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.

The one exception is noncovered range queries, which work well with clustered indexes, but may or may not be supported by nonclustered indexes, depending on the size of the range.

Consider 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.

Other Indexing Guidelines

Here are some other considerations for choosing indexes:

- If an index key is unique, be sure to define it as unique. Then, the optimizer knows immediately that only one row will be returned for 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. However, Adaptive Server does not automatically create an index on the referencing column. If your application updates and deletes primary keys, 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 12-6.
- 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. More rows fit on a page, speeding up leaf-level scans of nonclustered index pages. Also, more data fits on the distribution page for your index, increasing the accuracy of index statistics.
- Use small datatypes whenever it fits your design.
 - Numerics compare 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 Joins” on page 8-18 for more information.

Choosing Nonclustered Indexes

When you consider 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?
- Is there a lot of duplication?

Because of overhead, add nonclustered indexes only when your testing shows that they are helpful.

Choices include:

- Columns used for aggregates
- Columns used for joins or in **order by** or **group by** clauses

Performance Price for Data Modification

With each insert, all nonclustered indexes have to be updated, so there is a performance price to pay. The leaf level has one entry per row, so every insert into a table means one insert for each nonclustered index on the table.

All nonclustered indexes need to be updated:

- For each insert into the table.
- For each delete from the table.
- For any update to the table that changes any part of an index's key or that deletes a row from one page and inserts it on another page.
- For almost every update to the clustered index key. Usually, such an update means that the row is moved to a different page.
- For every data page split.

Choosing Composite Indexes

If your needs analysis shows that more than one column would make 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
- 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.

User Perceptions and Covered Queries

Covered queries can provide excellent response time for specific queries, although they may sometimes confuse users by providing much slower response time for very similar-looking queries. With the composite nonclustered index on *au_lname*, *au_fname*, *au_id*, this query runs very fast:

```
select au_id
  from authors
 where au_fname = "Eliot" and au_lname = "Wilk"
```

This covered point query needs to perform only three reads to find the value on the leaf-level row in the nonclustered index of a 5000-row table.

Users might not understand why this similar-looking query (using the same index) does not perform quite as well:

```
select au_fname, au_lname
  from authors
 where au_id = "A1714224678"
```

However, this query does not include the leading column of the index, so it has to scan the entire leaf level of the index, about 95 reads.

Adding a column to the select list, which may seem like a minor change to users, 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. But the optimizer has no way of knowing the number of duplicates in the third column (*au_id*) of a nonclustered index: it could match a single row, or it could match one-half of the rows in a table. A composite index can be used only when it covers the query or when the first column appears in the *where* clause.

Adding an unindexed column to a query that includes the leading column of the composite index adds only a single page read to this query, when it must read the data page to find the phone number:

```
select au_id, phone
  from authors
 where au_fname = "Eliot" and au_lname = "Wilk"
```

The Importance of Order in Composite Indexes

The preceding examples highlight the importance of the order of the columns in composite indexes. Table 7-5 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. The performance described in this table is for *where* clauses in the form:

```
where column_name = value
```

Table 7-5: Composite nonclustered index ordering and performance

Columns named in the <i>where</i> clause	Performance with only <i>au_lname</i> , <i>au_fname</i> and/or <i>au_id</i> in the select list	Performance with other columns in the select list
<i>au_lname</i> or <i>au_lname</i> , <i>au_fname</i> or <i>au_lname</i> , <i>au_fname</i> , <i>au_id</i>	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)
<i>au_fname</i> or <i>au_id</i> or <i>au_fname</i> , <i>au_id</i>	Moderate; index is scanned to return values	Poor; index not used, table scan

Choose the right ordering of the composite index so that most queries form a prefix subset.

Advantages of Composite Indexes

Composite indexes have these advantages:

- A dense composite index provides many opportunities for index covering.
- A composite index with qualifications on each of the keys will probably return fewer records than a query 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

Disadvantages of Composite Indexes

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 too wide because of long keys
- Composite indexes where only the second or third portion, or an even later portion, is used in the `where` clause

Key Size and Index Size

Small index entries yield small indexes, producing less index I/O to execute queries. Longer keys produce fewer entries per page, so an index requires more pages at each level and, in some cases, additional index levels.

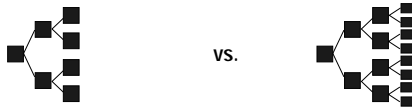
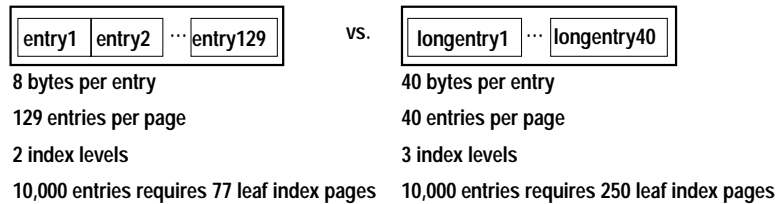


Figure 7-10: Sample rows for small and large index entries

The disadvantage of having a covering nonclustered index, particularly if the index entry is wide, is that there is a larger index to traverse, and updates are more costly.

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 1 or 2 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:

- 1,000,000 rows
- 10 rows per page; pages are 75 percent full, so the table has approximately 135,000 pages
- 190,000 (19 percent) 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 pages 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 would require reading 190,000 data pages, plus about 19 percent of the leaf level of the index, adding about 1,500 pages. Since a table scan requires only 135,000 pages, the nonclustered index would probably not be used.

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 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.

Here are four possible indexing strategies, identified by numbers used in subsequent discussion:

1. Nonclustered index on *titles(title)*; clustered index on *titles(price)*
2. Clustered index on *titles(title)*; nonclustered index on *titles(price)*
3. Nonclustered index on *titles(title, price)*
4. Nonclustered index on *titles(price, title)*; this was the best choice in the range query on price.

Table 7-6 shows some estimates of index sizes and I/O for the range query on *price* and the point query on *title*. 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 7-6: Comparing index strategies for two queries

Possible Index Choice	Index Pages	Range Query on <i>price</i>	Point Query on <i>title</i>
1 Nonclustered on <i>title</i> Clustered on <i>price</i>	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 <i>title</i> Nonclustered on <i>price</i>	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 <i>title, price</i>	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 <i>price, title</i>	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 Figure 7-6 shows that:

- For the range query on *price*, indexing 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, the nonclustered index in *price, title*, for range queries on *price*
- Choice 2, the clustered index on *title*, since it 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 these 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.

Index Statistics

When you create an index on a table that contains data, Adaptive Server creates a distribution page containing two kinds of statistics about index values:

- A distribution table, showing the distribution of keys in the index
- A density table, showing the density of key values

An index's distribution page is created when you create an index on a table that contains data. If you create an index on an empty table, no distribution page is created. If you truncate the table (removing all of its rows) the distribution page is dropped.

The data on the distribution page is not automatically maintained by Adaptive Server. You must run the `update statistics` command to update the data on the distribution page when the data in the table has changed since the distribution page was last update. Some activities that can cause this are:

- The distribution of the keys in an index can change due to rows being added or dropped. Indexes on ascending key values need to have statistics updated periodically so that new key values are included in the distribution table.
- Truncating a table and reloading the data, since truncating the table drops the distribution page.

If distribution page statistics are incorrect, query plans may be less than optimal.

The Distribution Table

The distribution table stores information about the distribution of key values in the index. The distribution table stores values only for the first key of a compound index.

The distribution table contains a list of key values called steps. The number of steps on the distribution page depends on the key size and whether the column stores variable-length data.

The statistics page looks very much like a data page or an index page, but there is one major difference—the statistics page does not contain a row offset table. The density table occupies this space on the page; for each key in the index, the density uses 2 bytes of storage. The rest of the page is available to store the steps. Figure 7-11 shows how to compute the number of steps that will be stored on the distribution page. Fixed-length columns have 2 bytes of overhead per step; variable-length columns have 7 bytes of overhead per step.

Fixed-Length Key

$$\text{Number of keys} = \frac{2016 - (\text{Number of keys} * 2)}{\text{Bytes per key} + 2}$$

Variable-Length Key

$$\text{Number of keys} = \frac{2016 - (\text{Number of keys} * 2)}{\text{Bytes per key} + 7}$$

Figure 7-11: Formulas for computing number of distribution page values

For variable-length columns, the defined (maximum) length is always used.

Once the number of steps is determined, Adaptive Server divides the number of rows in the table by the number of steps and then stores the data value for every *n*th row in the distribution table. Figure 7-12 shows this process for a small part of an index.

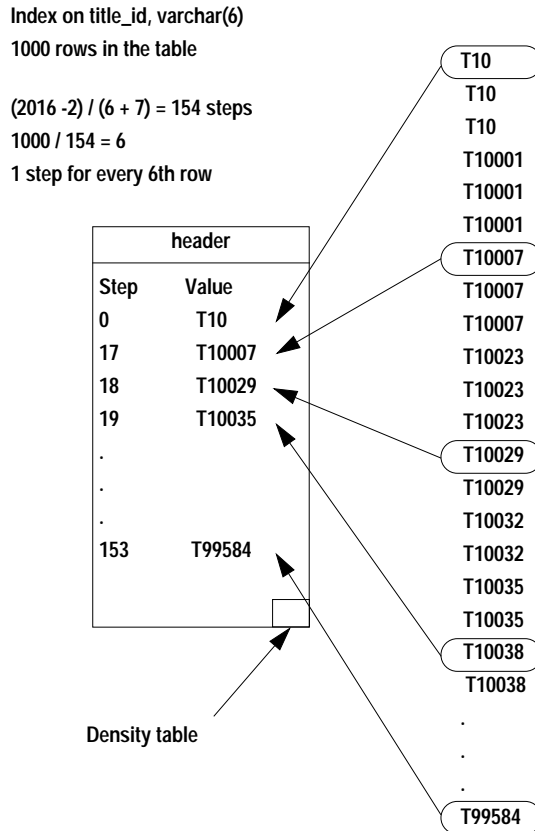


Figure 7-12: Building the distribution page

The Density Table

The query optimizer uses statistics to estimate the cost of using an index. One of these statistics is called the **density**. The density is the average proportion of duplicate keys in the index. It varies between 0 and 100 percent. An index with N rows whose keys are unique will have a density of $1/N$; an index whose keys are all duplicates of each other will have a density of 100 percent.

Since the index shown in Figure 7-12 is built on a single key, the density table contains just one entry. For indexes with multiple keys, the density table contains a value for each prefix of keys in the index. That is, for an index on columns A, B, C, D, it stores the density for:

- A
- A, B
- A, B, C
- A, B, C, D

If density statistics are not available, the optimizer uses default percentages, as shown in Table 7-7.

Table 7-7: Default density percentages

	Examples	Default
Equality	col = x	10%
Closed interval	col > x and col < y or col between x and y	25%
Open end range	col > x col >= x col < x col <= x	33%

For example, if no statistics page exists for an index on *authors(city)*, the optimizer estimates that this query would return 10 percent of the rows in the table for each key value:

```
select au_fname, au_lname, pub_name
   from authors a, publishers p
  where a.city = p.city
```

How the Optimizer Uses the Statistics

The optimizer uses index statistics to estimate the usefulness of an index and to decide join order. For example, this query finds a large number of titles between \$20 and \$30:

```
select title from titles
 where price between $20.00 and $30.00
```

However, this query finds only a few rows between \$1000 and \$1010:

```
select titles from titles
 where price between $1000.00 and $1010.00
```

The number of rows returned by these two queries may be different, and this affects the usefulness of nonclustered indexes. For a small number of rows, a nonclustered index can help reduce I/O, but for a large result set, a table scan is usually more efficient than nonclustered index access.

The statistics used by the optimizer include

- The number of rows in the table
- The number of pages for the table or on the leaf level of the nonclustered index
- The density value
- The distribution step values and the number of steps in the distribution table

How the Optimizer Uses the Distribution Table

The optimizer compares values given in the query to the values in the distribution table. Only one of the following conditions can be true for a given value:

- The value falls between two consecutive rows in the table.
- The value equals one row in the middle of the table.
- The value equals the first row or the last row in the table.
- The value equals more than one row in the middle of the table.
- The value equals more than one row, including the first or last row in the table.
- The value is less than the first row or greater than the last row in the table. (In this case, you should run `update statistics`.)

Depending on which condition is true, the optimizer uses formulas involving the step location (beginning, end, or middle of the page), the number of steps, the number of rows in the table, and the density to compute an estimated number of rows.

How the Optimizer Uses the Density Table

The optimizer uses the density table to help estimate the number of rows that a query will return. Even if the value of a search argument is not known when the query is optimized, the optimizer can use the density values in an index, as long as the leading column(s) are specified for composite indexes.

Index Maintenance

Indexes should evolve as your system evolves. To ensure this:

- Monitor queries to determine if indexes are still appropriate for your applications.
- Drop and rebuild indexes only if they are hurting performance.
- Keep index statistics up to date.

Monitoring Applications and Indexes Over Time

Periodically, check the query plans, as described in Chapter 9, “Understanding Query Plans,” and the I/O statistics for your most frequent user queries. Pay special attention to nonclustered indexes that support range queries. They are most likely to switch to table scans if the data changes. Indexes should be based on the transactions and processes that are being run, not on the original database design

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. Use query plans to determine whether your indexes are being used.

Maintaining Index Statistics

The distribution page for an index is not maintained as data rows are added and deleted. The database owner must issue an `update statistics` command to ensure that statistics are current. The syntax is:

```
update statistics table_name [index_name]
```

For example, this command updates all the indexes on the *authors* table:

```
update statistics authors
```

To update a single index, give the table name and index name:

```
update statistics titles titles_idx
```

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 an increasing key value. Date columns, such as those in a sales entry application, 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 are match a given value.

► Note

Failure to update statistics can severely hurt performance.

Adaptive Server is a very efficient transaction processing engine. However, to perform well, it relies heavily on distribution page data. If statistics are not up to date, a query that should take only a few seconds could may much longer.

Rebuilding Indexes

Rebuilding indexes reclaims space in the B-trees. As pages are split and rows are deleted, indexes may contain many pages that are only half full or 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.

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

- Space usage exceeds estimates because heavy data modification has left many data and index pages partially full.
- `dbcc` has identified errors in the index.

If you rebuild a clustered index, all nonclustered indexes are re-created, since creating the clustered index moves rows to different pages. Nonclustered indexes must be re-created to point to the correct pages.

Rebuilding indexes takes time. Use the `sp_estspace` procedure to estimate the time needed to generate indexes. See the sample output in “Index Entries Are Too Large” on page 7-4.

In most 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
- To create a new group of indexes to help generate a set of reports

See “Creating Indexes” on page 23-3 for information about configuration parameters that increase the speed of creating indexes.

Speeding Index Creation with *sorted_data*

If data is already sorted, using the `sorted_data` option for the `create index` command can save index creation time. This option can be used for both clustered and nonclustered indexes. See “Creating an Index on Sorted Data” on page 23-4 for more information.

Displaying Information About Indexes

The *sysindexes* table in each database contains one row for each:

- Index
- Heap table
- Table that contains text or image columns

In each row, it stores pointers that provide the starting point for various types of data access. These pointers are shown in Table 7-8.

The contents are maintained as index pages are allocated and deallocated.

Table 7-8: Page pointers for unpartitioned tables in the sysindexes table

Object Type	<i>indid</i>	<i>root</i>	<i>first</i>
Heap table	0	Last data page in the table's data chain	First data page in the table's data chain
Clustered index	1	Root page of the index	First data page in the table's data chain
Nonclustered index	2-250	Root page of the index	First leaf page of the nonclustered index
Text/image object	255	First page of the object	First page of the object

The *sysindexes.distribution* column displays 0 if no statistics exist for an index, because the index was created on an empty table. If *distribution* is 0, the value points to the distribution page.

The *doampg* and *ioampg* columns in *sysindexes* store pointers to the first OAM page for the data pages (*doampg*) or index pages (*ioampg*). The system functions *data_pgs*, *reserved_pgs*, and *used_pgs* use these pointers and the object ID to provide information about space usage. See “Determining or Estimating the Sizes of Tables and Indexes” on page 6-1 for a sample query.

Tips and Tricks for Indexes

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 23-3 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 is usually worth the effort.

Keeping Index Entries Short and Avoiding Overhead

Avoid storing purely numeric IDs as character data (*varchar*, *char*, or *nvarchar*). Use integer or numeric IDs whenever possible to:

- Save storage space on the data pages
- Make index entries more compact
- Allow more rows on the distribution page, if the ID is used as an index key
- 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 and to increase the number of distribution page 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 7-45.

Choosing Fillfactors for Indexes

By default, Adaptive Server creates indexes that are completely full at the leaf level and leaves room for two rows on the intermediate pages for growth. The *fillfactor* option for the *create index* command allows you to specify how full to create index pages and the data

pages of clustered indexes. Figure 7-13 illustrates a table that has a fillfactor of 50 percent.

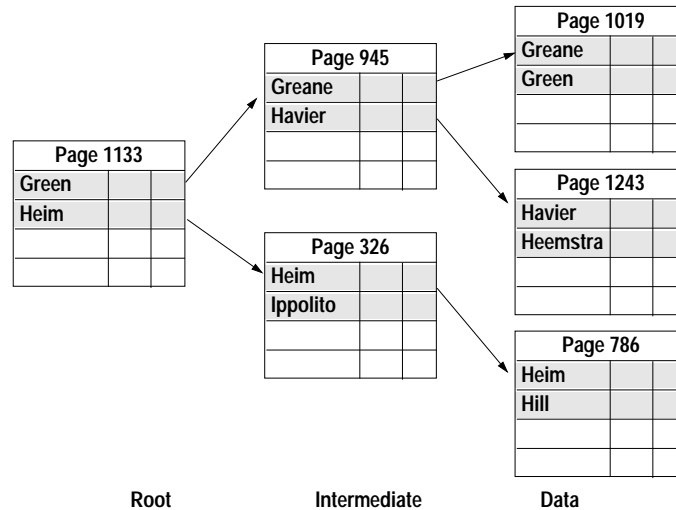


Figure 7-13: Table and clustered index with fillfactor set to 50 percent

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*. Note that the *fillfactor* is used only when you create the index; it is not maintained over time. The purpose of *fillfactor* is to provide a performance boost for tables that will experience growth; maintaining that *fillfactor* by continuing to split partially full pages would defeat the purpose.

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.

Advantages of Using *fillfactor*

Setting *fillfactor* to a low value provides a temporary performance enhancement. Its benefits fade away as inserts to the database increase the amount of space used on data pages. A lower *fillfactor* provides these benefits:

- It reduces page splits.
- It can reduce lock contention, 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 a set of eight pages on an extent are likely to be read sequentially.

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* only to help reduce the performance impact of lock contention on index rows, you may want to use `max_rows_per_page` instead. 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.

Using *sp_sysmon* to Observe the Effects of Changing *fillfactor*

`sp_sysmon` generates output that allows you to observe how different *fillfactor* values affect system performance. Pay particular attention to the performance categories in the output that are most likely to be affected by changes in *fillfactor*: page splits and lock contention.

See Chapter 24, "Monitoring Performance with `sp_sysmon`," and the topics "Page Splits" on page 24-51 and "Lock Management" on page 24-58 in that chapter.

Adaptive Server Monitor, a separate Sybase product, can pinpoint where problems are at the object level.

8

Understanding the Query Optimizer

This chapter introduces the Adaptive Server query optimizer and explains how different types of queries are optimized.

This chapter contains the following sections:

- What Is Query Optimization? 8-1
- Adaptive Server's Cost-Based Optimizer 8-2
- Diagnostic Tools for Query Optimization 8-6
- Optimizer Strategies 8-8
- Search Arguments and Using Indexes 8-9
- Optimizing Joins 8-13
- Optimization of `or` clauses and `in (values_list)` 8-23
- Optimizing Aggregates 8-26
- Optimizing Subqueries 8-28
- Update Operations 8-34

This chapter covers the optimization techniques used for serial queries. Chapter 14, "Parallel Query Optimization," discusses additional optimization techniques used for parallel queries.

What Is Query Optimization?

Query optimization is the process of analyzing individual queries to determine what resources they use and whether the use of resources can be reduced. For any query, you need to understand how it accesses database objects, the sizes of the objects, and the indexes on the tables in order to determine whether it is possible to improve the query's performance. These topics are addressed in Chapters 2–7.

Symptoms of Optimization Problems

Some symptoms of optimization problems are:

- A query runs more slowly than you expect, based on indexes and table size.
- A query runs more slowly than similar queries.
- A query suddenly starts running more slowly than usual.

- A query processed within a stored procedure takes longer than when it is processed as an ad hoc statement.
- The query plan shows the use of a table scan when you expect it to use an index.

Sources of Optimization Problems

Some sources of optimization problems are:

- Statistics have not been updated recently, so the actual data distribution does not match the values used by Adaptive Server to optimize queries.
- The *where* clause causes the optimizer to select an inappropriate strategy.
- The rows to be referenced by a given transaction do not fit the pattern reflected by the index statistics.
- An index is being used to access a large portion of the table.
- No appropriate index exists for a critical query.
- A stored procedure was compiled before significant changes to the underlying tables were performed.

Adaptive Server's Cost-Based Optimizer

The optimizer is the part of Adaptive Server's code that examines parsed and normalized queries and information about database objects. The input to the optimizer is a parsed SQL query; the output from the optimizer is a **query plan**. The query plan is the ordered set of steps required to carry out the query, including the methods (table scan, index choice, and so on) to access each table. A query plan is compiled code that is ready to run. Figure 8-1 shows query processing steps and optimizer inputs.

The Adaptive Server optimizer finds the best query plan—the plan that is least the costly in terms of time. For many Transact-SQL queries, there are many possible query plans. The optimizer reviews

all possible plans and estimates the cost of each plan. Adaptive Server selects the least costly plan, and compiles and executes it.

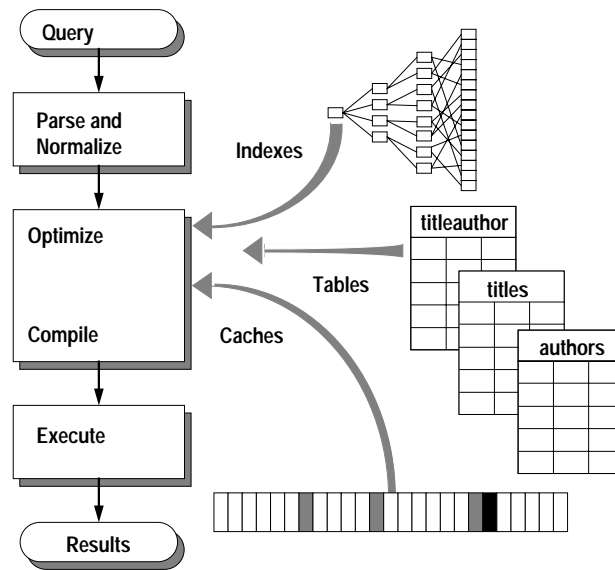


Figure 8-1: Query execution steps

Steps in Query Processing

When you execute a Transact-SQL query, Adaptive Server processes it in these steps:

1. The query is parsed and normalized. The parser ensures that the SQL syntax is correct. Normalization ensures that all the objects referenced in the query exist.
2. The query is optimized. Each part of the query is analyzed, and the best query plan is chosen. Optimization includes the following:
 - Each table is analyzed
 - Cost of each index is estimated
 - Join order is chosen
 - Final access method is determined
3. The chosen query plan is compiled.

4. The query is executed, and the results are returned to the user.

Working with the Optimizer

The goal of the optimizer is to select the access method that reduces the total time needed to process a query. The optimizer bases its choice on the contents of the tables being queried and on other factors such as cache strategies, cache size, and I/O size. Since disk access is generally the most expensive operation, the most important task in optimizing queries is to provide the optimizer with appropriate index choices, based on the transactions to be performed.

Adaptive Server's cost-based query optimizer has evolved over many years, taking into account many different issues. However, because of its general-purpose nature, the optimizer may select a query plan that is different from the one you expect. In certain situations, it may make the incorrect choice of access methods. In some cases, this may be the result of inaccurate or incomplete information. In other cases, additional analysis and the use of special query processing options can determine the source of the problem and provide solutions or workarounds. Chapter 10, "Advanced Optimizing Techniques," describes additional tools for debugging problems like this.

How Is "Fast" Determined?

Knowing what the optimizer considers to be fast and slow can significantly improve your understanding of the query plan chosen by the optimizer. The significant costs in query processing are:

- Physical I/Os, when pages must be read from disk
- Logical I/Os, when pages in cache must be read repeatedly for a query

For queries with an `order by` clause or a `distinct` clause, the optimizer adds the physical and logical I/O cost for performing sorts to the cost of the physical and logical I/O to read data and index pages.

The optimizer assigns 18 as the cost of a physical I/O and 2 as the cost of a logical I/O. Some operations using worktables use 5 as the cost of writing to the worktable. These are relative units of cost and do not represent time units such as milliseconds or ticks.

Query Optimization and Plans

Query plans consist of retrieval tactics and an ordered set of execution steps to retrieve the data needed by the query. In developing query plans, the optimizer examines:

- The size of each table in the query, both in rows and data pages.
- The size of the available data cache, the size of I/O supported by the cache, and the cache strategy to be used.
- The indexes, and the types of indexes, that exist on the tables and columns used in the query.
- Whether the index covers the query, that is, whether the query can be satisfied by retrieving data from index keys without having to access the data pages. Adaptive Server can use indexes that cover queries, even if no *where* clauses are included in the query.
- The density and distribution of keys in the indexes. Adaptive Server maintains statistics for index keys. See “Index Statistics” on page 7-42 for more information on index statistics.
- The estimated cost of physical and logical reads and cost of caching.
- Optimizable join clauses and the best join order, considering the costs and number of table scans required for each join and the usefulness of indexes in limiting the scans.
- Whether building a worktable (an internal, temporary table) with an index on the join columns would be faster than repeated table scans if there are no useful indexes for the inner table in a join.
- Whether the query contains a *max* or *min* aggregate that can use an index to find the value without scanning the table.
- Whether the pages will be needed repeatedly to satisfy a query such as a join or whether a fetch-and-discard strategy can be employed because the pages need to be scanned only once.

For each plan, the optimizer determines the total cost in milliseconds. Adaptive Server then uses the best plan.

Stored procedures and triggers are optimized when the object is first executed, and the query plan is stored in cache. If other users execute the same procedure while an unused copy of the plan resides in cache, the compiled query plan is copied into cache, rather than being recompiled.

Diagnostic Tools for Query Optimization

Adaptive Server provides the following diagnostic tools for query optimization:

- `set showplan on` displays the steps performed for each query in a batch. It is often used with `set noexec on`, especially for queries that return large numbers of rows.
- `set statistics io on` displays the number of logical and physical reads and writes required by the query. If resource limits are enabled, it also displays the total actual I/O. This tool is described in Chapter 7, “Indexing for Performance.”
- `set statistics subquerycache on` displays the number of cache hits and misses and the number of rows in the cache for each subquery. See “Displaying Subquery Cache Information” on page 8-33 for examples.
- `set statistics time on` displays the time it takes to parse and compile each command. It displays the time it takes to execute each step of the query. The “parse and compile” and “execution” times are given in `timeticks`, the exact value of which is machine-dependent. The “elapsed time” and “cpu time” are given in milliseconds. See “Using `set statistics time`” on page 8-7 for more information.
- `dbcc traceon (302)` provides additional information about why particular plans were chosen and is often used when the optimizer chooses plans that seem incorrect.

You can use many of these options at the same time, but some of them suppress others, as described below.

`showplan`, `statistics io`, and other commands produce their output while stored procedures are being run. The system procedures that you might use for checking table structure or indexes as you test optimization strategies can produce voluminous output. You may want to have hard copies of your table schemas and index information, or you can use separate windows for running system procedures such as `sp_helpindex`.

For lengthy queries and batches, you may want to save `showplan` and `statistics io` output in files. The “echo input” flag to `isql` echoes the input into the output file, with line numbers included. The syntax is:

```
isql -P password -e -i input_file -o outputfile
```

Using *showplan* and *noexec* Together

showplan is often used in conjunction with *set noexec on*, which prevents the SQL statements from being executed. Be sure to issue the *showplan* command, or any other set commands, before you issue the *noexec* command. Once you issue *set noexec on*, the only command that Adaptive Server executes is *set noexec off*. This example shows the correct order:

```
set showplan on
set noexec on
go
select au_lname, au_fname
      from authors
      where au_id = "A137406537"
go
```

noexec and *statistics io*

While *showplan* and *noexec* make useful companions, *noexec* stops all the output of *statistics io*. The *statistics io* command reports actual disk I/O; while *noexec* is in effect, no I/O takes place, so the reports are not printed.

Using *set statistics time*

set statistics time displays information about the time it takes to execute Adaptive Server commands. It prints these statistics:

- Parse and compile time – the number of CPU ticks taken to parse, optimize, and compile the query
- Execution time – the number of CPU ticks taken to execute the query
- Adaptive Server CPU time – the number of CPU ticks taken to execute the query, converted to milliseconds

To see the *clock_rate* for your system, execute:

```
sp_configure "sql server clock tick length"
```

See “sql server clock tick length” on page 11-122 of the *System Administration Guide* for more information.

- Adaptive Server elapsed time – the difference in milliseconds between the time the command started and the current time, as taken from the operating system clock

The formula in Figure 8-2 converts ticks to milliseconds:

$$\text{Milliseconds} = \frac{\text{CPU_ticks} * \text{clock_rate}}{1000}$$

Figure 8-2: Formula for converting ticks to milliseconds

```
select type, sum(advance)
from titles t, titleauthor ta, authors a,
publishers p
where t.title_id = ta.title_id
      and ta.au_id = a.au_id
      and p.pub_id = t.pub_id
      and (a.state = "NY" or a.state = "CA")
      and p.state != "NY"
group by type
having max(total_sales) > 100000
```

The following output shows that the query was parsed and compiled in one clock tick, or 100 ms. It took 120 ticks, or 12,000 ms., to execute. Total elapsed time was 17,843 ms., indicating that Adaptive Server spent some time processing other tasks or waiting for disk or network I/O to complete.

```
Parse and Compile Time 1.
SQL Server cpu time: 100 ms.
type
```

```
-----
UNDECIDED                210,500.00
business                 256,000.00
cooking                  286,500.00
news                     266,000.00
```

```
Execution Time 120.
SQL Server cpu time: 12000 ms. SQL Server elapsed time: 17843 ms.
```

Optimizer Strategies

The following sections explain how the optimizer analyzes these types of queries:

- Search arguments in the where clause
- Joins
- Queries using or clauses and the in (*values_list*) predicate
- Aggregates

- Subqueries
- Updates

Search Arguments and Using Indexes

It is important to distinguish between *where* and *having* clause specifications that can be used as search arguments to find query results via indexes and those that are used later in query processing. This distinction creates a finer definition for the search argument. Search arguments, or SARGs, can be used to determine an access path to the data rows. They match index keys and determine which indexes are used to locate and retrieve the matching data rows. Other selection criteria are additional qualifications that are applied to the rows after they have been located.

Consider this query, with an index on *au_lname*, *au_fname*:

```
select au_lname, au_fname, phone
  from authors
  where au_lname = "Gerland"
        and city = "San Francisco"
```

The clause:

```
au_lname = "Gerland"
```

qualifies as a SARG because:

- There is an index on *au_lname*
- There are no functions or other operations on the column name
- The operator is a valid SARG operator
- The datatype of the constant matches the datatype of the column

The clause:

```
city = "San Francisco"
```

matches all the criteria above except the first—there is no index on *city*. There is no index on the *city* column. In this case, the index on *au_lname* would probably be used as the search argument for the query. All pages with a matching author name are brought into cache, and each matching row is examined to see if the city also matches.

One of the first steps the optimizer performs is to separate the SARGs from other qualifications on the query so that it can cost the access methods for each SARG.

SARGs in *where* and *having* Clauses

The optimizer looks for SARGs in the *where* and *having* clauses of a query and for indexes that match the columns. If your query uses one or more of these clauses to scan an index, you will see the *showplan* output “Keys are: <keylist>” immediately following the index name information. If you think your query should be using an index, and it causes table scans instead, look carefully at the search clauses and operators.

Indexable Search Argument Syntax

Indexable search arguments are expressions in one of these forms:

```
<column> <operator> <expression>  
<expression> <operator> <column>  
<column> is null
```

Where:

- *column* is only a column name. Functions, expressions, or concatenation added to the column name always require a table scan.
- *operator* must be one of the following:
=, >, <, >=, <=, !>, !<, <>, !=, is null.
- *expression* is a constant or an expression that evaluates to a constant. The optimizer uses the index statistics differently, depending on whether the value of the expression is a constant or an expression:
 - If *expression* is a constant, its value is known when the query is optimized. It can be used by the optimizer to look up distribution values in the index statistics.
 - If the value of *expression* is not known at compile time, the optimizer uses the density from the distribution page to estimate the number of rows to be returned by the query. The value of variables, mathematical expressions, concatenation, and functions cannot be known until the query is executed.

The non-equality operators, <> and !=, are special cases. The optimizer can check for covering nonclustered indexes on the column name is indexed and can choose a nonmatching index scan, if an index covers the query.

Search Argument Equivalents

The optimizer looks for equivalents that it can convert to SARGs. These are listed in Table 8-1.

Table 8-1: SARG equivalents

Clause	Conversion
<code>between</code>	Converted to <code>>=</code> and <code><=</code> clauses.
<code>like</code>	If the first character in the pattern is a constant, <code>like</code> clauses can be converted to greater than or less than queries. For example, <code>like "sm%"</code> becomes <code>>= "sm" and < "sn"</code> . The expression <code>like "%x"</code> cannot be optimized.
<i>expression</i>	If the <i>expression</i> portion of the <code>where</code> clause contains arithmetic expressions that can be converted to a constant, the optimizer can use the density values, and may use the index, but it cannot use the distribution table on the index.

The following are some examples of optimizable search arguments:

```
au_lname = "Bennett"
price >= $12.00
advance > 10000 and advance < 20000
au_lname like "Ben%" and price > $12.00
```

These search arguments are optimizable, but they use only the density value, not the distribution values, from the index statistics:

```
salary = 12 * 3000
price = @value
```

The following search arguments are **not** optimizable:

```
salary = commission /*both are column names*/
advance * 2 = 5000 /*expression on column side
not permitted */

advance = $10000
or price = $20.00 /*see "OR strategy" */

substring(au_lname,1,3) = "Ben" /* no functions on
column name */
```

Transitive Closure Applied to SARGs

The optimizer applies transitive closure to search arguments. For example, the following query joins *titles* and *titleauthor* on *title_id* and includes a SARG on *titles.title_id*:

```
select au_lname, title
from titles t, titleauthor ta, authors a
where t.title_id = ta.title_id
      and a.au_id = ta.au_id
      and t.title_id = "T81002"
```

This query is optimized as if it also included the SARG on *titleauthor.title_id*:

```
select au_lname, title
from titles t, titleauthor ta, authors a
where t.title_id = ta.title_id
      and a.au_id = ta.au_id
      and t.title_id = "T81002"
      and ta.title_id = "T81002"
```

Transitive closure is applied only to SARGs. It is not applied to joins.

Guidelines for Creating Search Arguments

Follow these guidelines when you write search arguments for your queries:

- Avoid functions, arithmetic operations, and other expressions on the column side of search clauses.
- Avoid incompatible datatypes.
- Use the leading column of a composite index. The optimization of secondary keys provides less performance.
- Use all the search arguments you can to give the optimizer as much as possible to work with.
- If a query has over 128 predicates for a table, put the most potentially useful clauses near the beginning of the query, since only the first 128 SARGs on each table are used during optimization. (All of the search conditions, however, are used to qualify the rows.)
- Check `showplan` output to see which keys and indexes are used.

Figure 8-3 shows how predicates are applied by the optimizer and during query execution, and questions to ask when examining predicates and index choices.

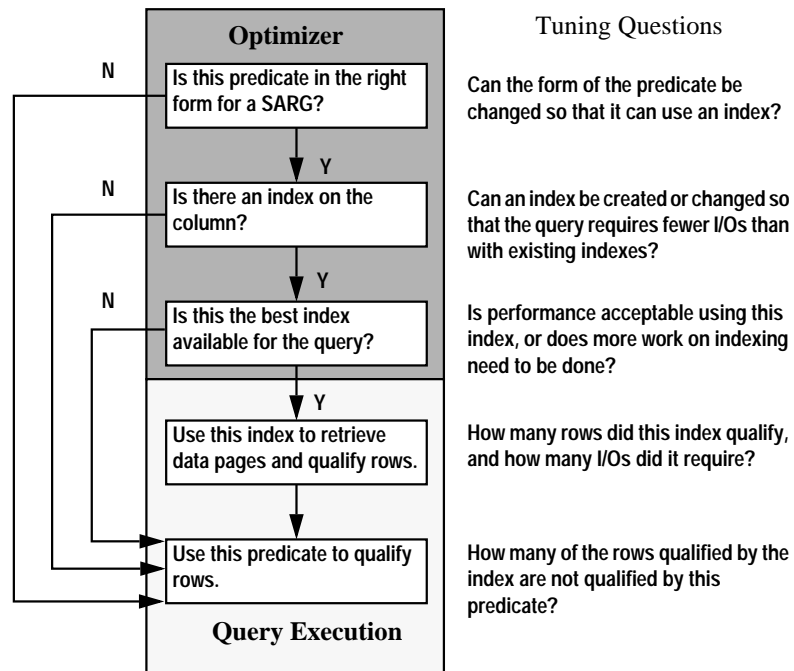


Figure 8-3: SARGs and index choices

Optimizing Joins

Joins pull information from two or more tables. This requires **nested iterations** on the tables involved. In a two-table join, one table is treated as the outer table; the other table becomes the inner table. Adaptive Server examines the outer table for rows that satisfy the query conditions. For each row in the outer table that qualifies, Adaptive Server then examines the inner table, looking at each row where the join columns match.

In `showplan` output, the order of "FROM TABLE" messages indicates the order in which Adaptive Server chooses to join tables. See "FROM TABLE Message" on page 9-4 for an example that joins three tables.

Optimizing the join columns in queries is extremely important. Relational databases make extremely heavy use of joins. Queries that

perform joins on several tables are especially critical, as explained in the following sections.

Some subqueries are also converted to joins. See “Flattening in, any, and exists Subqueries” on page 8-28.

Join Syntax

Join clauses take the form:

```
table1.column_name <operator> table2.column_name
```

The join operators are:

```
=, >, >=, <, <=, !>, !<, !=, <>, *=, =*
```

How Joins Are Processed

When the optimizer creates a query plan for a join query:

- It determines which index to use for each table by estimating the I/O required for each possible index and for a table scan. The least costly method is selected.
- If no useful index exists on the inner table of a join, the optimizer may decide to build a temporary index, a process called **reformatting**. See “Saving I/O Using the Reformatting Strategy” on page 8-17.
- It determines the join order, basing the decision on the total cost estimates for the possible join orders.
- It determines the I/O size and caching strategy. If an unindexed table is small, the optimizer may decide to read the entire table into cache.

Factors such as indexes and density of keys, which determine costs on single-table selects, become much more critical for joins.

Basic Join Processing

The process of creating the result set for a join is to nest the tables, and to scan the inner tables repeatedly for each qualifying row in the outer table, as shown in Figure 8-4.

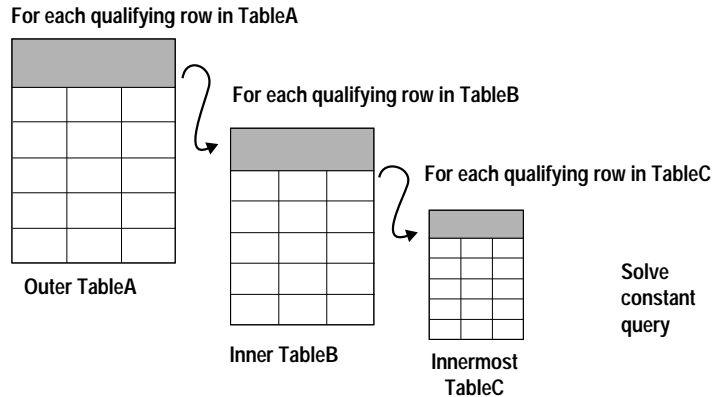


Figure 8-4: Nesting of tables during a join

In Figure 8-4, the access to the tables to be joined is nested:

- *TableA* is accessed once.
- *TableB* is accessed once for each qualifying row in *TableA*.
- *TableC* is accessed once for each qualifying row in *TableB* **each time** that *TableB* is accessed.

For example, if 15 rows from *TableA* match the conditions in the query, *TableB* is accessed 15 times. If 20 rows from *TableB* match for each matching row in *TableA*, then *TableC* is scanned 300 times. If *TableC* is small, or has a useful index, the I/O count stays reasonably small. If *TableC* is large and unindexed, the optimizer may choose to use the reformatting strategy to avoid performing extensive I/O.

Choice of Inner and Outer Tables

The outer table is usually the one that has:

- The smallest number of qualifying rows, and/or
- The largest numbers of reads required to locate rows.

The inner table usually has:

- The largest number of qualifying rows, and/or
- The smallest number of reads required to locate rows.

For example, when you join a large, unindexed table to a smaller table with indexes on the join key, the optimizer chooses:

- The large table as the outer table. Adaptive Server will have to read this large table only once.
- The indexed table as the inner table. Each time Adaptive Server needs to access the inner table, it will take only a few reads to find rows.

Figure 8-5 shows a large, unindexed table and a small, indexed table.

```
select TableA.colx, TableB.coly
from TableA, TableB
where TableA.col1 = TableB.col1
      and TableB.col2 = "anything"
      and TableA.col2 = "something"
```

Table A:
1,000,000 rows
10 rows per page
100,000 pages
No index

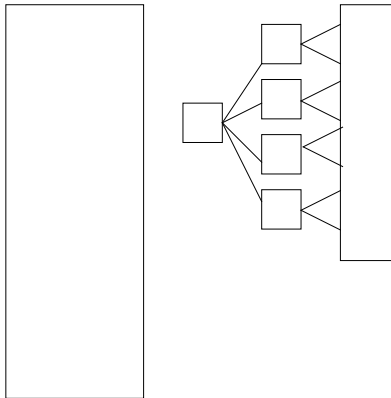


Table B:
100,000 rows
10 rows per page
10,000 pages
Clustered index on join column

Figure 8-5: Alternate join orders and page reads

If *TableA* is the outer table, it is accessed via a table scan. When the first qualifying row is found, the clustered index on *TableB* is used to find the row or rows where *TableB.col1* matches the value retrieved from *TableA*. Then, the scan on *TableA* continues until another match is found. The clustered index is used again to retrieve the next set of matching rows from *TableB*. This continues until *TableA* has been

completely scanned. If 10 rows from *TableA* match the search criteria, the number of page reads required for the query is:

	Pages Read
Table scan of <i>TableA</i>	100,000
10 clustered index accesses of <i>TableB</i> +	30
Total	100,030

If *TableB* is the outer table, the clustered index is used to find the first row that matches the search criteria. Then, *TableA* is scanned to find the rows where *TableA.col1* matches the value retrieved from *TableB*. When the table scan completes, another row is read from the data pages for *TableB*, and *TableA* is scanned again. This continues until all matching rows have been retrieved from *TableB*. If 10 rows in *TableB* match, this access choice would require the following number of page reads:

	Pages Read
1 clustered index access of <i>TableB</i> +	3
10 table scans of <i>TableA</i>	1,000,000
Total	1,000,003

Saving I/O Using the Reformatting Strategy

Adding another large, unindexed table to the query in Figure 8-5 would create a huge volume of required page reads. If the new table also contained 100,000 pages, for example, and contained 20 qualifying rows, *TableA* would need to be scanned 20 times, at a cost of 100,000 reads each time. The optimizer costs this plan, but also costs a process called **reformatting**. Adaptive Server can create a temporary clustered index on the join column for the inner table.

The steps in the reformatting strategy are:

- Creating a worktable
- Inserting the needed columns from the qualifying rows
- Creating a clustered index on the join columns of the worktable
- Using the clustered index in the join to retrieve the qualifying rows from each table

The main cost of the reformatting strategy is the time and I/O necessary to create the worktable and to build the clustered index on the worktable. Adaptive Server uses reformatting only when the reformatting cost is less than the cost of joining the tables by repeatedly performing a table scan on the inner table.

A `showplan` message indicates when Adaptive Server is using the reformatting strategy and includes other messages showing the steps used to build the worktables. See “Reformatting Message” on page 9-35.

Index Density and Joins

For any join using an index, the optimizer uses a statistic called the **density** to help optimize the query. The density is the average proportion of duplicate keys in the index. It varies between 0 percent and 100 percent. An index whose keys are all duplicates of each other will have a density of 100 percent; an index with N rows, whose keys are all unique, will have a density of $1/N$.

The query optimizer uses the density to estimate the number of rows that will be returned for each scan of the inner table of a join for a particular index. For example, if the optimizer is considering a join with a 10,000-row table, and an index on the table has a density of 25 percent, the optimizer will estimate 2500 rows per scan for a join using that index.

Adaptive Server maintains a density for each prefix of columns in a composite index. That is, it keeps a density on the first column, the first and second columns, the first, second, and third columns, and so on, up to and including the entire set of columns in the index. The optimizer uses the appropriate density for an index when estimating the cost of a join using that index. In a 10,000-row table with an index on seven columns, the entire seven-column key might have a density of $1/10,000$, while the first column might have a density of only $1/2$, indicating that it would return 5000 rows.

The density statistics on an index are maintained by the `create index` and `update statistics` commands.

If statistics are not available, the optimizer uses default percentages, as shown in Table 8-2:

Table 8-2: Default density percentages

	Examples	Default
Equality	col = x	10%
Closed interval	col > x and col < y col >= x and col <= y col between x and y	25%
Open end range	col > x col >= x col < x col <= x	33%

For example, if there is no statistics page for an index on *authors(city)*, the optimizer estimates that 10 percent of the rows must be returned for this query:

```
select au_fname, au_lname, pub_name
   from authors a, publishers p
  where a.city = p.city
```

Datatype Mismatches and Joins

One of the most common problems in optimizing joins on tables that have indexes is that the datatypes of the join columns are incompatible. When this occurs, one of the datatypes must be implicitly converted to the other, using Adaptive Server's datatype hierarchy. Datatypes that are lower in the hierarchy are always converted to higher types.

Some examples where problems frequently arise are:

- Joins between *char not null* with *char null* or *varchar*. A *char* datatype that allows null values is stored internally as a *varchar*.
- Joins using numeric datatypes such as *int* and *float*. Allowing null values is not a problem with numeric datatypes in joins.

To avoid these problems, make sure that the datatypes on join columns are the same when you create the tables. Use either *null* or *not null* for two columns with the *char* datatype. Match precision and scale for numeric types. See "Joins and Datatypes" on page 11-6 for more information and for workarounds for existing tables.

Join Permutations

When you are joining four or fewer tables, Adaptive Server considers all possible permutations of join orders for the tables. It establishes this cutoff because the number of permutations of join orders multiplies with each additional table, requiring lengthy computation time for large joins. For joins of more than 4 tables, it examines joins orders for four tables at a time.

The method the optimizer uses to determine join order has excellent results for most queries and requires much less CPU time than examining all permutations of all combinations. The `set table count` command allows you to specify the number of tables that the optimizer considers at one time, up to a maximum of 8. See “Increasing the Number of Tables Considered by the Optimizer” on page 10-8 for more information.

Joins in Queries with More Than Four Tables

Changing the order of the tables in the `from` clause usually has no effect on the query plan, even on tables that join more than four tables.

When you have more than four tables in the `from` clause, Adaptive Server optimizes each subset of four tables. Then, it remembers the outer table from the best plan involving four tables, eliminates it from the set of tables in the `from` clause, and optimizes the best set of four tables out of the remaining tables. It continues until only four tables remain, at which point it optimizes those four tables normally.

For example, suppose you have a `select` statement with the following `from` clause:

```
from T1, T2, T3, T4, T5, T6
```

The optimizer looks at all possible sets of 4 tables taken from these 6 tables. The 15 possible combinations of all 6 tables are:

T1, T2, T3, T4
 T1, T2, T3, T5
 T1, T2, T3, T6
 T1, T2, T4, T5
 T1, T2, T4, T6
 T1, T2, T5, T6
 T1, T3, T4, T5
 T1, T3, T4, T6
 T1, T3, T5, T6
 T1, T4, T5, T6
 T2, T3, T4, T5
 T2, T3, T4, T6
 T2, T3, T5, T6
 T2, T4, T5, T6
 T3, T4, T5, T6

For each one of these combinations, the optimizer looks at all the join orders (permutations). For each set of 4 tables, there are 24 possible join orders, for a total of 360 (24 * 15) permutations. For example, for the set of tables *T2*, *T3*, *T5*, and *T6*, the optimizer looks at these 24 possible orders:

T2, T3, T5, T6
 T2, T3, T6, T5
 T2, T5, T3, T6
 T2, T5, T6, T3
 T2, T6, T3, T5
 T2, T6, T5, T3
 T3, T2, T5, T6
 T3, T2, T6, T5
 T3, T5, T2, T6
 T3, T5, T6, T2
 T3, T6, T2, T5
 T3, T6, T5, T2
 T5, T2, T3, T6
 T5, T2, T6, T3
 T5, T3, T2, T6
 T5, T3, T6, T2
 T5, T6, T2, T3
 T5, T6, T3, T2
 T6, T2, T3, T5
 T6, T2, T5, T3
 T6, T3, T2, T5
 T6, T3, T5, T2
 T6, T5, T2, T3
 T6, T5, T3, T2

Let's say that the best join order is:

T5, T3, T6, T2

At this point, *T5* is designated as the outermost table in the query.

The next step is to choose the second-outermost table. The optimizer eliminates *T5* from consideration as it chooses the rest of the join order. Now, it has to determine where *T1*, *T2*, *T3*, *T4*, and *T6* fit into the rest of the join order. It looks at all the combinations of four tables chosen from these five:

T1, T2, T3, T4
T1, T2, T3, T6
T1, T2, T4, T6
T1, T3, T4, T6
T2, T3, T4, T6

It looks at all the join orders for each of these combinations, remembering that *T5* is the outermost table in the join. Let's say that the best order in which to join the remaining tables to *T5* is *T3*, *T6*, *T2*, and *T4*.

So the optimizer chooses *T3* as the next table after *T5* in the join order for the entire query. It eliminates *T3* from consideration in choosing the rest of the join order.

The remaining tables are:

T1, T2, T4, T6

Now we're down to four tables, so the optimizer looks at all the join orders for all the remaining tables. Let's say the best join order is:

T6, T2, T4, T1

This means that the join order for the entire query is:

T5, T3, T6, T2, T4, T1

Even though the optimizer looks at the join orders for only four tables at a time, the fact that it does this for all combinations of four tables that appear in the *from* clause makes the order of tables in the *from* clause irrelevant.

The only time that the order of tables in the *from* clause can make any difference is when the optimizer comes up with the same cost estimate for two join orders. In that case, it chooses the first of the two join orders that it encounters. The order of tables in the *from* clause affects the order in which the optimizer evaluates the join orders, so in this one case, it can have an effect on the query plan. Notice that it does not have an effect on the query cost or on the query performance.

Optimization of *or* clauses and *in (values_list)*

The choice of access methods for queries that contain *or* clauses or an *in (values_list)* clause depends on the indexes that exist on the columns named in these clauses, and whether it is possible for the set of clauses to result in duplicate values.

or syntax

or clauses take one of the following forms:

```
where column_name1 = <value>
       or column_name1 = <value>
```

or:

```
where column_name1 = <value>
       or column_name2 = <value>
```

in (values_list) Converts to *or* Processing

The parser converts *in* lists to *or* clauses, so this query:

```
select title_id, price
       from titles
       where title_id in ("PS1372", "PS2091","PS2106")
```

becomes:

```
select title_id, price
       from titles
       where title_id = "PS1372"
              or title_id = "PS2091"
              or title_id = "PS2106"
```

How *or* Clauses Are Processed

A query using *or* clauses is a union of more than one query. Although some rows may match more than one of the conditions, each row is returned only once.

If any of the columns used in an *or* clause or the column in the *in* clause are not indexed, the query must use a table scan. If indexes exist on all of the columns, the optimizer chooses one of two strategies:

- Multiple matching index scans

- A special strategy called the **OR strategy**

or Clauses and Table Scans

A query with or clauses or an in (values_list) uses a table scan if either of these conditions is true:

- The cost of all the index accesses is greater than the cost of a table scan, or
- At least one of the clauses names a column that is not indexed, so the only way to resolve the clause is to perform a table scan.

If the query performs a table scan, all the conditions are applied to each row, as the pages are scanned.

Multiple Matching Index Scans

Adaptive Server uses multiple matching index scans when there is no possibility that the or clauses will return duplicate rows. The example in Figure 8-6 on page 8-25 shows two or clauses, with a row that satisfies both conditions. This query must be resolved by the OR strategy.

On the other hand, this query cannot return any duplicate rows:

```
select title
  from titles
  where title_id in ("T6650", "T95065", "T11365")
```

This query can be resolved using multiple matching index scans.

The optimizer determines which index to use for each or clause or value in the in (values_list) clause by costing each clause or value separately. If each column named in a clause is indexed, a different index can be used for each clause or value.

If the query performs a multiple matching index scan, the query uses the appropriate index for each or clause or value in the in list, and returns the rows to the user, as the data rows are accessed.

The OR Strategy

If the query must use the special OR strategy because the query could return duplicate rows, the query uses the appropriate index, but first it retrieves all the **row IDs** for rows that satisfy each or clause and stores them in a worktable in *tempdb*. Adaptive Server then sorts the worktable to remove the duplicate row IDs. In *showplan* output, this worktable is called a **dynamic index**. The row IDs are used to retrieve the rows from the base tables.

Figure 8-6 illustrates the process of building and sorting a dynamic index for this query:

```
select title_id, price
  from titles
 where price < $15
    or title like "Compute%"
```

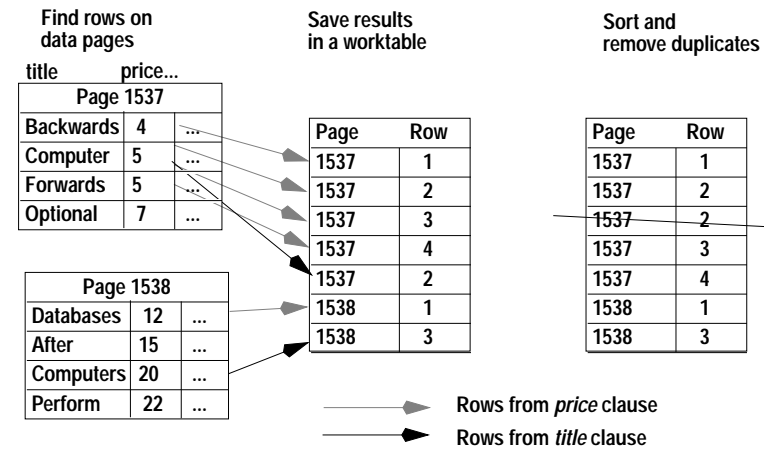


Figure 8-6: Resolving or queries

The optimizer estimates the cost index access for each clause in the query. For queries with or clauses on different columns in the same table, the optimizer can choose to use a different index for each clause. The query uses a table scan if either of these conditions is true:

- The cost of all the index accesses is greater than the cost of a table scan, or
- At least one of the clauses names a column that is not indexed, so the only way to resolve the clause is to perform a table scan.

Queries in cursors cannot use the OR strategy, but must perform a table scan. However, queries in cursors can use the multiple matching index scans strategy.

Locking and the OR Strategy

During a select operation, the OR strategy maintains a shared lock on all the pages accessed during the entire operation. No other process

can update the rows, since row IDs cannot be allowed to change until the rows are returned to the user. For long sets of `OR` clauses or in (*values_list*) sets, this can affect concurrency, since it limits access by other users. Be especially careful of queries that use the `OR` strategy if you are using isolation level 3 or the `holdlock` clause.

Optimizing Aggregates

Aggregates are processed in two steps:

- First, appropriate indexes are used to retrieve the appropriate rows, or the table is scanned. For vector (grouped) aggregates, the results are placed in a worktable. For scalar aggregates, results are computed in a variable in memory.
- Second, the worktable is scanned to return the results for vector aggregates, or the results are returned from the internal variable.

In many cases, vector aggregates can be optimized to use a composite nonclustered index on the aggregated column and the grouping column, if any, rather than performing table scans. For example, if the *titles* table has a nonclustered index on *type*, *price*, the following query retrieves its results from the leaf level of the nonclustered index:

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

Both vector and scalar aggregates can use covering nonclustered indexes to reduce I/O. For example, the following query can use the index on *type*, *price*:

```
select min(price)
  from titles
  where type = "business"
```


Table 8-3 shows access methods that the optimizer can choose for queries with aggregates when there is no *where*, *having* or *group by* clause in the query.

Table 8-3: Special access methods for aggregates

Aggregate	Index Type	Access Method
<i>min</i>	Any index	Use first value of the root page of the index
<i>max</i>	Clustered index	Follow last pointer on root page and intermediate pages to data page, uses last value
	Nonclustered index	Follow last pointer on root page and intermediate pages to leaf page, uses last value
<i>count(*)</i>	Nonclustered index	Count all rows in the leaf level of the index with the smallest number of pages.
<i>count(col_name)</i>	Covering nonclustered index	Count all non-null values in the leaf level of the smallest index containing the column name.

Combining *max* and *min* Aggregates

When used separately, *max* and *min* aggregates on indexed columns use special processing if there is no *where* clause in the query:

- *min* aggregates retrieve the first value on the root page of the index, performing a single read to find the value.
- *max* aggregates follow the last entry on the last page at each index level until they reach the leaf level. For a clustered index, the number of reads required is the height of the index tree, plus one read for the data page. For a nonclustered index, the number of reads required is the height of the index tree.

However, when *min* and *max* are used together, this optimization is not available. The entire leaf level of a nonclustered index is scanned to pick up the first and last values. If no nonclustered index includes the value, a table scan is used. For more discussion and a workaround, see “Aggregates” on page 11-5.

Optimizing Subqueries

► **Note**

This section describes the mode of subquery processing that was introduced in SQL Server release 11.0. If your stored procedures, triggers, and views were created prior to release 11.0, and they have not been dropped and re-created, they may not use the same processing. See `sp_procmode` in the *Adaptive Server Reference Manual* for more information on determining the processing mode.

Subqueries use the following special optimizations to improve performance:

- Flattening – converting the subquery to a join
- Materializing – storing the subquery results in a worktable
- Short circuiting – placing the subquery last in the execution order
- Caching subquery results – recording the results of executions

The following sections explain these strategies. See “showplan Messages for Subqueries” on page 9-45 for an explanation of the `showplan` messages for subquery processing.

Flattening *in*, *any*, and *exists* Subqueries

Adaptive Server can flatten some quantified predicate subqueries (those introduced with *in*, *any*, or *exists*) to an **existence join**. Instead of the usual nested iteration through a table that returns all matching values, an existence join returns TRUE when it finds the first matching value and then stops processing. If no matching value is found, it returns FALSE.

A subquery introduced with *in*, *any*, or *exists* is flattened to an existence join unless one of the following is true:

- The subquery is correlated and contains one or more aggregates, or
- The outer query also uses *or*.

All *in*, *any*, and *exists* queries test for the existence of qualifying values and return TRUE as soon as a matching row is found.

Existence joins can be optimized as effectively as regular joins. The major difference is that existence joins stop looking as soon as they find the first match.

For example, the optimizer converts the following subquery to an existence join:

```
select title
  from titles
 where title_id in
       (select title_id
        from titleauthor)
 and title like "A Tutorial%"
```

The join query looks like the following ordinary join, although it does not return the same results:

```
select title
  from titles T, titleauthor TA
 where T.title_id = TA.title_id
 and title like "A Tutorial%"
```

In the *pubtune* database, two books match the search string on *title*. Each book has multiple authors, so it has multiple entries in *titleauthor*. A regular join returns five rows, but the subquery returns only two rows, one for each *title_id*, since it stops execution of the join at the first matching row.

Flattening Expression Subqueries

Expression subqueries are subqueries that are included in a query's select list or that are introduced by >, >=, <, <=, =, or !=. Adaptive Server converts, or **flattens**, expression subqueries to **equijoins** if:

- The subquery joins on unique columns or returns unique columns, and
- There is a unique index on the columns.

Materializing Subquery Results

In some cases, the subquery is processed in two steps: the results from the inner query are **materialized**, or stored in a temporary worktable, before the outer query is executed. The subquery is executed in one step, and the results of this execution are stored and then used in a second step. Adaptive Server materializes these types of subqueries:

- Noncorrelated expression subqueries
- Quantified predicate subqueries containing aggregates where the **having** clause includes the correlation condition

Noncorrelated Expression Subqueries

Noncorrelated expression subqueries must return a single value.

When a subquery is not correlated, it returns the same value, regardless of the row being processed in the outer query. The execution steps are:

- Adaptive Server executes the subquery and stores the result in an internal variable.
- Adaptive Server substitutes the result value for the subquery in the outer query.

The following query contains a noncorrelated expression subquery:

```
select title_id
from titles
where total_sales = (select max(total_sales)
                    from ts_temp)
```

Adaptive Server transforms the query to:

```
select <internal_variable> = max(total_sales)
   from ts_temp

select title_id
   from titles
  where total_sales = <internal_variable>
```

The search clause in the second step of this transformation can be optimized. If there is an index on *total_sales*, the query can use it.

Quantified Predicate Subqueries Containing Aggregates

Some subqueries that contain vector (grouped) aggregates can be materialized. These are:

- Noncorrelated quantified predicate subqueries
- Correlated quantified predicate subqueries correlated only in the **having** clause

The materialization of the subquery results in these two steps:

- Adaptive Server executes the subquery first and stores the results in a worktable.

- Adaptive Server joins the outer table to the worktable as an existence join. In most cases, this join cannot be optimized because statistics for the worktable are not available.

Materialization saves the cost of evaluating the aggregates once for each row in the table. For example, this query:

```
select title_id
from titles
where total_sales in (select max(total_sales)
                      from titles
                      group by type)
```

Executes in these steps:

```
select maxsales = max(total_sales)
into #work
from titles
group by type

select title_id
from titles, #work
where total_sales = maxsales
```

Short Circuiting

When there are **where** clauses in addition to a subquery, Adaptive Server executes the subquery or subqueries last to avoid unnecessary executions of the subqueries. Depending on the clauses in the query, it is often possible to avoid executing the subquery because other clauses have already determined whether the row is to be returned:

- If any **and** clauses evaluate to **FALSE**, the row will not be returned.
- If any **or** clauses evaluate to **TRUE**, the row will be returned.

In both of these cases, as soon as the status of the row is determined by the evaluation of one clause, no other clauses need to be applied to that row.

Subquery Introduced with an *and* Clause

When **and** joins the clauses, the evaluation of the list stops as soon as any clause evaluates to **FALSE**.

This query contains two **and** clauses, in addition to the subquery:

```

select au_fname, au_lname, title, royaltyper
from titles t, authors a, titleauthor ta
where t.title_id = ta.title_id
and a.au_id = ta.au_id
and advance >= (select avg(advance)
                 from titles t2
                 where t2.type = t.type)

and price > 100
and au_ord = 1

```

Adaptive Server orders the execution steps to evaluate the subquery last. If a row does not meet an `and` condition, Adaptive Server discards the row without checking any more `and` conditions and begins to evaluate the next row, so the subquery is not processed unless the row meets all of the `and` conditions.

Subquery Introduced with an `or` Clause

If a query's `where` conditions are connected by `or`, evaluation stops when any clause evaluates to `TRUE`.

This query contains two `and` clauses in addition to the subquery:

```

select au_fname, au_lname, title
from titles t, authors a, titleauthor ta
where t.title_id = ta.title_id
and a.au_id = ta.au_id
and (advance > (select avg(advance)
                 from titles t2
                 where t.type = t2.type)
     or title = "Best laid plans"
     or price > $100)

```

Again, Adaptive Server reorders the query to evaluate the subquery last. If a row meets the condition of the `or` clause, Adaptive Server does not process the subquery and proceeds to evaluate the next row.

Subquery Results Caching

When it cannot flatten or materialize a subquery, Adaptive Server uses an in-memory cache to store the results of each evaluation of the subquery. The lookup key for the subquery cache is:

- The values in the correlation columns, plus
- The join column for quantified subqueries.

While the query runs, Adaptive Server tracks the number of times a needed subquery result is found in cache. This is called a **cache hit**. If

the cache hit ratio is high, it means that the cache is reducing the number of times that the subquery executes. If the cache hit ratio is low, the cache is not useful, and it is reduced in size as the query runs.

Caching the subquery results improves performance when there are duplicate values in the join columns or the correlation columns. It is even more effective when the values are ordered, as in a query that uses an index. Caching does not help performance when there are no duplicate correlation values.

Displaying Subquery Cache Information

The set statistics subquerycache on command displays the number of cache hits and misses and the number of rows in the cache for each subquery. The following example shows subquery cache statistics:

```
set statistics subquerycache on

select type, title_id
from titles
where price > all
      (select price
       from titles
       where advance < 15000)
```

```
Statement: 1 Subquery: 1 cache size: 75 hits: 4925 misses: 75
```

If the statement includes subqueries on either side of a union, the subqueries are numbered sequentially through both sides of the union. For example:

```
select id from sysobjects a
where id = 1 and id =
      (select max(id)
       from sysobjects b where a.id = b.id)
union
select id from sysobjects a
where id = 1 and id =
      (select max(id)
       from sysobjects b where a.id = b.id)
```

```
Statement: 1 Subquery: 1 cache size: 1 hits: 0 misses: 1
```

```
Statement: 1 Subquery: 2 cache size: 1 hits: 0 misses: 1
```

Optimizing Subqueries

When queries containing subqueries are not flattened or materialized:

- The outer query and each unflattened subquery is optimized one at a time
- The innermost subqueries (the most deeply nested) are optimized first.
- The estimated buffer cache usage for each subquery is propagated outward to help evaluate the I/O cost and strategy of the outer queries.

In many queries that contain subqueries, a subquery is “attached” to one of the outer table scans by a two-step process. First, the optimizer finds the point in the join order where all the correlation columns are available. Then, the optimizer searches from that point to find the table access that qualifies the fewest rows and attaches the subquery to that table. The subquery is then executed for each qualifying row from the table it is attached to.

Update Operations

Adaptive Server handles updates in different ways, depending on the changes being made to the data and the indexes used to locate the rows. The two major types of updates are **deferred updates** and **direct updates**. Adaptive Server performs direct updates whenever possible.

Direct Updates

Adaptive Server performs direct updates in a single pass, as follows:

- It locates the affected index and data rows
- It writes the log records for the changes to the transaction log
- It makes the changes to the data pages and any affected index pages

There are three techniques for performing direct updates:

- In-place updates
- Cheap direct updates
- Expensive direct updates

Direct updates require less overhead than deferred updates and are generally faster, as they limit the number of log scans, reduce logging, save traversal of index B-trees (reducing lock contention),

and save I/O because Adaptive Server does not have to refetch pages to perform modifications based on log records.

In-Place Updates

Adaptive Server performs in-place updates whenever possible.

When Adaptive Server performs an in-place update, subsequent rows on the page are not moved; the row IDs remain the same and the pointers in the row offset table are not changed.

For an in-place update, the following requirements must be met:

- The row being changed must not change its length.
- The column being updated cannot be the key, or part of the key, of a clustered index. Because the rows in a clustered index are stored in key order, a change to the key almost always means that the row location is changed.
- One or more indexes must be unique or must allow duplicates.
- The update statement does not include a join.
- The affected columns are not used for referential integrity.
- There cannot be a trigger on the column.
- The table cannot be replicated (via Replication Server).

Figure 8-7 shows an in-place update. The *pubdate* column is a fixed-length column, so the length of the data row is not changed. The access method in this example could be a table scan or a clustered or nonclustered index on *title_id*.

```
update titles set pubdate = "Jun 30 1988"
where title_id = "BU1032"
```

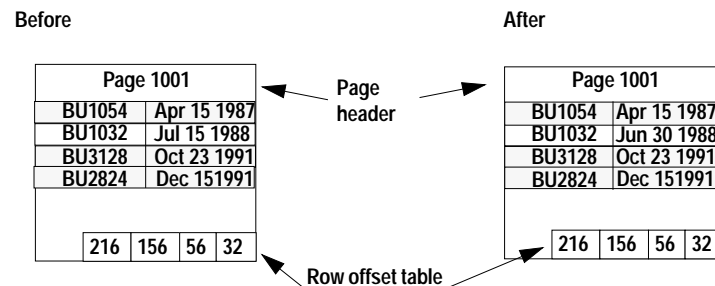


Figure 8-7: In-place update

An in-place update is the fastest type of update because it makes a single change to the data page. It changes all affected index entries by deleting the old index rows and inserting the new index row. In-place updates affect only indexes whose keys are changed by the update, since the page and row locations are not changed.

Cheap Direct Updates

If Adaptive Server cannot perform an update in place, it tries to perform a cheap direct update—changing the row and rewriting it at the same offset on the page. Subsequent rows on the page are moved up or down so that the data remains contiguous on the page, but the row IDs remain the same. The pointers in the row offset table changes to reflect the new locations.

For a cheap direct update, the following requirements must be met:

- The length of the data in the row is changed, but the row still fits on the same data page, or the row length is not changed, but there is a trigger on the table or the table is replicated.
- The column being updated cannot be the key, or part of the key, of a clustered index. Because Adaptive Server stores the rows of a clustered index in key order, a change to the key almost always means that the row location is changed.
- One or more indexes must be unique or must allow duplicates.
- The update statement does not include a join.
- The affected columns are not used for referential integrity.

The update in Figure 8-8 changes the length of the second row from 20 to 100 bytes. Other rows are moved down on the page, so the row

offset values are increased for the rows that follow the updated row on the page.

```
update titles set title = "Coping with Computer Stress in
the Modern Electronic Work Environment"
where title_id = "BU1032"
```

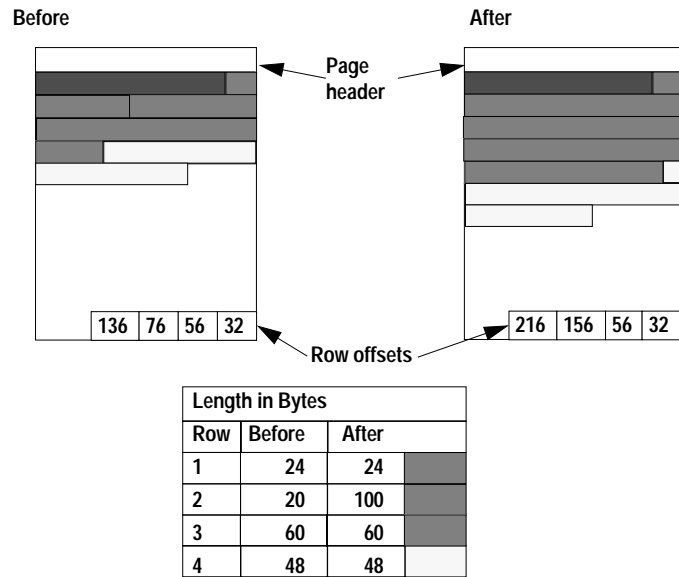


Figure 8-8: Cheap direct update

Cheap direct updates are almost as fast as in-place updates. They require the same amount of I/O, but slightly more processing. Two changes are made to the data page (the row and the offset table). Any changed index keys are updated by deleting old values and inserting new values. Cheap direct updates affect only indexes whose keys are changed by the update, since the page and row ID are not changed.

Expensive Direct Updates

If the data does not fit on the same page, Adaptive Server performs an expensive direct update, if possible. An expensive direct update deletes the data row, including all index entries, and then inserts the modified row and index entries.

Adaptive Server uses a table scan or an index to find the row in its original location and then deletes the row. If the table has a clustered

Deferred Updates

Adaptive Server uses deferred updates when direct update conditions are not met. A deferred update is the slowest type of update.

In a deferred update, Adaptive Server:

- Locates the affected data rows, writing the log records for deferred delete and insert of the data pages as rows are located.
- Reads the log records for the transaction and performs the deletes on the data pages and any affected index rows.
- Read the log records a second time, and performs all inserts on the data pages, and inserts any affected index rows.

Deferred updates are always required for:

- Updates that use joins
- Updates to columns used for referential integrity

Deferred updates are also required when:

- The update moves a row to a new page while the table is being accessed via a table scan or a clustered index.
- Duplicate rows are not allowed in the table, and there is no unique index to prevent them.
- The index used to find the data row is not unique, and the row is moved because the update changes the clustered index key or because the new row does not fit on the page.

Deferred updates incur more overhead than direct updates because they require Adaptive Server to re-read the transaction log to make the final changes to the data and indexes. This involves additional traversal of the index trees.

For example, if there is a clustered index on *title*, this query performs a deferred update:

```
update titles set title = "Portable C Software"
where title = "Designing Portable Software"
```

Deferred Index Inserts

Adaptive Server performs deferred index updates when the update affects the index used in the query or when the update affects columns in a unique index. In this type of update, Adaptive Server:

- Deletes the index entries in direct mode
- Updates the data page in direct mode, writing the deferred insert records for the index
- Reads the log records for the transaction and inserts the new values in the index in deferred mode

Deferred index insert mode must be used when the update changes the index used to find the row or when the update affects a unique index. A query must update a single, qualifying row only once—deferred index update mode ensures that a row is found only once during the index scan and that the query does not prematurely violate a uniqueness constraint.

The update in Figure 8-10 changes only the last name, but the index row is moved from one page to the next. To perform the update, Adaptive Server:

1. Reads index page 1133, deletes the index row for “Greene” from that page, and logs a deferred index scan record.
2. Changes “Green” to “Hubbard” on the data page in direct mode and continues the index scan to see if more rows need to be updated.
3. Inserts the new index row for “Hubbard” on page 1127.

Figure 8-10 shows the index and data pages prior to the deferred update operation, and the sequence in which the deferred update changes the data and index pages.

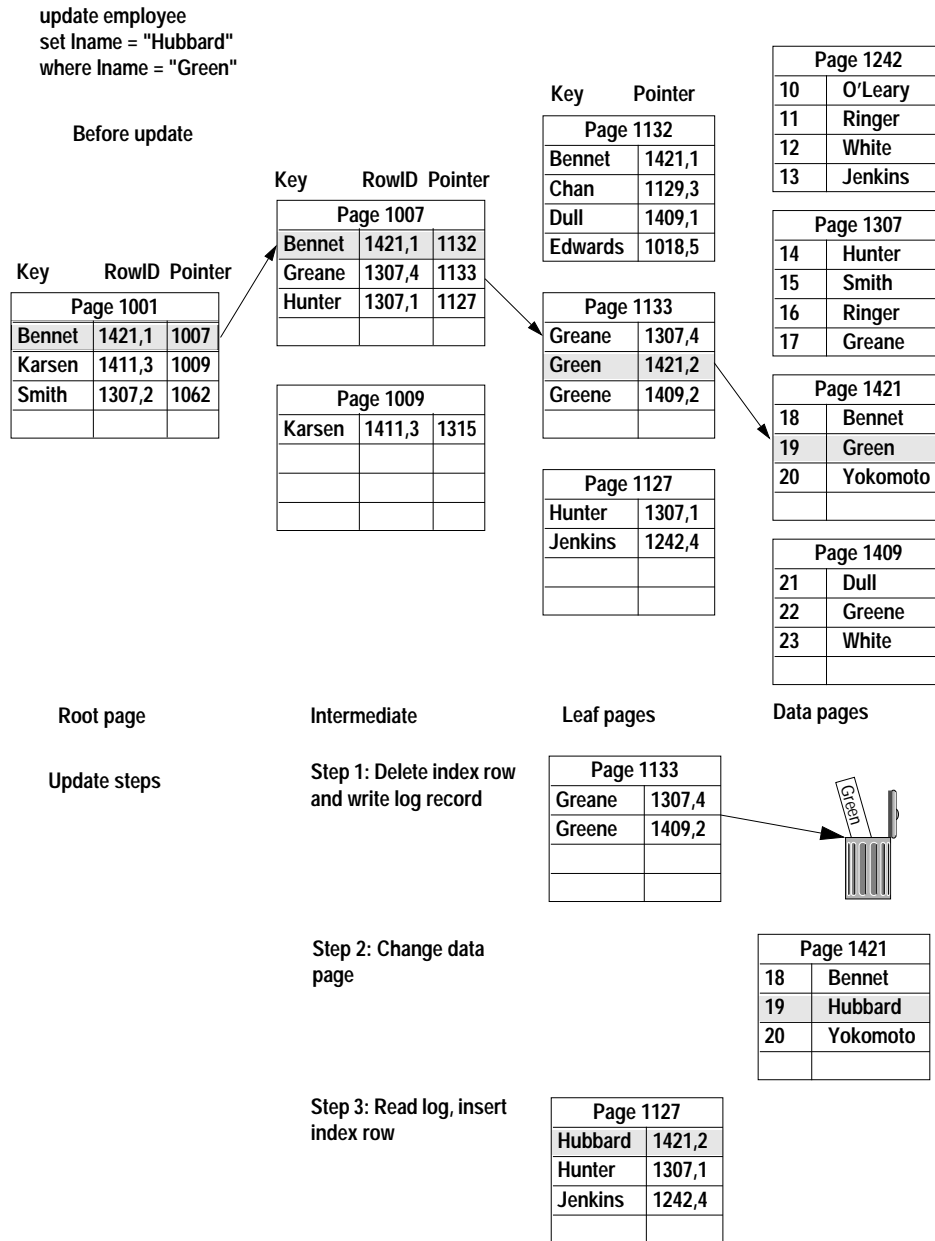


Figure 8-10: Deferred index update

Assume a similar update to the *titles* table:

```
update titles
set title = "Computer Phobic's Manual",
  advance = advance * 2
where title like "Computer Phob%"
```

This query shows a potential problem. If a scan of the nonclustered index on the *title* column found “Computer Phobia Manual,” changed the title, and multiplied the advance by 2, and then found the new index row “Computer Phobic’s Manual” and multiplied the advance by 2, the author might be quite delighted with the results, but the publishers would not!

Because similar problems arise with updates in joins, where a single row can match the join criteria more than once, join updates are always processed in deferred mode.

A deferred index delete may be faster than an expensive direct update, or it may be substantially slower, depending on the number of log records that need to be scanned and whether the log pages are still in cache.

Optimizing Updates

`showplan` messages provide information about whether an update will be performed in direct mode or deferred mode. If a direct update is not possible, Adaptive Server updates the data row in deferred mode. There are times when the optimizer cannot know whether a direct update or a deferred update will be performed, so two `showplan` messages are provided:

- The “deferred_varcol” message shows that the update may change the length of the row because a variable-length column is being updated. If the updated row fits on the page, the update is performed in direct mode; if the update does not fit on the page, the update is performed in deferred mode.
- The “deferred_index” message indicates that the changes to the data pages and the deletes to the index pages are performed in direct mode, but the inserts to the index pages are performed in deferred mode.

The different types of direct updates depend on information that is available only at run time. For example, the page actually has to be fetched and examined in order to determine whether the row fits on the page.

When you design and code your applications, be aware of the differences that can cause deferred updates. Follow these guidelines to help avoid deferred updates:

- Create at least one unique index on the table to encourage more direct updates.
- Whenever possible, use non-key columns in the `where` clause when updating a different key.
- If you do not use null values in your columns, declare them as `not null` in your `create table` statement.

Effects of Update Types and Indexes on Update Modes

Table 8-4 shows the effects of indexes and update types on update mode for three different types of updates. In all cases, duplicate rows are not allowed. For the indexed cases, the index is on `title_id`. The three types of updates are:

- A variable-length key column:

```
update titles set title_id = value
  where title_id = "T1234"
```
- A variable-length nonkey column:

```
update titles set date = value
  where title_id = "T1234"
```
- A fixed-length nonkey column:

```
update titles set notes = value
  where title_id = "T1234"
```

Figure 8-4 shows how a unique index can promote a more efficient update mode than a nonunique index on the same key. Pay particular attention to the differences between direct and deferred in the shaded areas of the table. For example, with a unique clustered index, all of these updates can be performed in direct mode, but they must be performed in deferred mode if the index is nonunique.

For a table with a nonunique clustered index, a unique index on any other column in the table provides improved update performance. In some cases, you may want to add an `IDENTITY` column to a table in

order to include the column as a key in an index that would otherwise be nonunique.

Table 8-4: Effects of indexing on update mode

Index	Update To:		
	Variable-Length Key	Fixed-Length Column	Variable-Length Column
No index	N/A	direct	deferred_varcol
Clustered, unique	direct	direct	direct
Clustered, not unique	deferred	deferred	deferred
Clustered, not unique, with a unique index on another column	deferred	direct	deferred_varcol
Nonclustered, unique	deferred_varcol	direct	direct
Nonclustered, not unique	deferred_varcol	direct	deferred_varcol

If the key for a table is fixed length, the only difference in update modes from those shown in the table occurs for nonclustered indexes. For a nonclustered, nonunique index, the update mode is `deferred_index` for updates to the key. For a nonclustered, unique index, the update mode is `direct` for updates to the key.

Choosing Fixed-Length Datatypes for Direct Updates

If the length of *varchar* or *varbinary* is close to the maximum length, use *char* or *binary* instead. Each variable-length column adds row overhead and increases the possibility of deferred updates.

Using *max_rows_per_page* to Increase Direct Updates

Using *max_rows_per_page* to reduce the number of rows allowed on a page increases direct updates, because an update that increases the length of a variable-length column may still fit on the same page. For more information on using *max_rows_per_page*, see Chapter 5, “Decreasing the Number of Rows per Page.”

Using *sp_sysmon* While Tuning Updates

You can use `showplan` to determine whether an update is deferred or direct, but `showplan` does not give you detailed information about the type of deferred or direct update. Output from the system procedure `sp_sysmon` or Adaptive Server Monitor supplies detailed statistics about the types of updates performed during a sample interval.

Run `sp_sysmon` as you tune updates, and look for reduced numbers of deferred updates, reduced locking, and reduced I/O. See “Updates” on page 24-42 and “Deletes” on page 24-42 for more information.

9

Understanding Query Plans

This chapter describes each message printed by the `showplan` utility. The `showplan` utility displays the steps performed for each query in a batch, the keys and indexes used for the query, the order of joins, and special optimizer strategies.

This chapter contains the following sections:

- Using `showplan` 9-1
- Basic `showplan` Messages 9-2
- `showplan` Messages for Query Clauses 9-12
- `showplan` Messages Describing Access Methods, Caching, and I/O Cost 9-23
- `showplan` Messages for Parallel Queries 9-39
- `showplan` Messages for Subqueries 9-45

Using `showplan`

To see the query plan for a query, use the following command:

```
set showplan on
```

To stop displaying query plans, use the following command:

```
set showplan off
```

Combining `showplan` and `noexec`

You can use `showplan` in conjunction with `set noexec on` to prevent SQL statements from being executed. Be sure to issue the `showplan` command, or any other `set` command before you issue the `noexec` command. Once you issue `set noexec on`, the only command that Adaptive Server executes is `set noexec off`. Both commands should be in a separate batch from the query you are analyzing. This example shows both `set` commands followed by a query:

```

set showplan on
set noexec on
go
select au_lname, au_fname
      from authors
      where au_id = "A1374065371"
go

```

If you need to create or drop indexes, remember that `set noexec on` also suppresses execution of these commands for the session in which you issue them.

More Tips on Using *showplan*

`showplan`, `statistics io`, and other commands produce their output while stored procedures are executed. You may want to have hard copies of your table schemas and index information. Or you can use separate windows for running system procedures such as `sp_helpindex` and for creating and dropping indexes.

Basic *showplan* Messages

This section describes `showplan` messages that are printed for most `select`, `insert`, `update`, and `delete` operations.

Table 9-1: Basic `showplan` messages

Message	Explanation	See
Query Plan for Statement <i>N</i> (at line <i>N</i>).	"Statement <i>N</i> " is the statement number within the batch; "line <i>N</i> " is the line number within the batch.	Page 9-3
STEP <i>N</i>	Each step for each statement is numbered sequentially. Numbers are restarted at 1 on each side of a <code>union</code> .	Page 9-3
The type of query is <i>query type</i> .	<i>query type</i> is replaced by the type of query: <code>SELECT</code> , <code>UPDATE</code> , <code>INSERT</code> , or any Transact-SQL statement type.	Page 9-4
FROM TABLE	Each occurrence of <code>FROM TABLE</code> indicates a table that will be read. The table name is listed on the next line. Table 9-3 on page 9-23 shows the access method messages that <code>showplan</code> displays for each table access.	Page 9-4

Table 9-1: Basic showplan messages (continued)

Message	Explanation	See
TO TABLE	Included when a command creates a worktable and for <code>insert...select</code> commands.	Page 9-7
Nested iteration.	Indicates the execution of a data retrieval loop.	Page 9-8
The update mode is direct. The update mode is deferred. The update mode is deferred_varcol. The update mode is deferred_index.	These messages indicate whether an <code>insert</code> , <code>delete</code> , or <code>update</code> is performed in direct or deferred update mode. See "Update Mode Messages" on page 9-9.	Page 9-9

Query Plan Delimiter Message

Query Plan for Statement *N* (at line *N*)

Adaptive Server prints this line once for each query in a batch. Its main function is to provide a visual cue that separates one clump of `showplan` output from the next clump. Line numbers are provided to help you match query output with your input.

Step Message

STEP *N*

`showplan` output displays "STEP *N*" for every query, where *N* is an integer, beginning with "STEP 1". For some queries, Adaptive Server cannot effectively retrieve the results in a single step and must break the query plan into several steps. For example, if a query includes a `group by` clause, Adaptive Server breaks it into at least two steps:

- One step to select the qualifying rows from the table and to group them, placing the results in a worktable
- Another step to return the rows from the worktable

This example demonstrates a single-step query.

```
select au_lname, au_fname
from authors
where city = "Oakland"
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is SELECT.

FROM TABLE

authors

Nested iteration.

Table Scan.

Ascending scan.

Positioning at start of table.

Using I/O Size 2 Kbytes.

With LRU Buffer Replacement Strategy.

Multiple-step queries are demonstrated in the first two examples following “GROUP BY Message” on page 9-13.

Query Type Message

The type of query is *query type*.

This message describes the type of query for each step. For most queries that require tuning, the value for *query type* is SELECT, INSERT, UPDATE, or DELETE. However, the *query type* can include any Transact-SQL command that you issue while *showplan* is enabled. For example, here is output from a *create index* command:

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is CREATE INDEX.

TO TABLE

titleauthor

FROM TABLE Message

FROM TABLE

tablename [

correlation]

This message indicates which table the query is reading from. The “FROM TABLE” message is followed on the next line by the table name. If the from clause includes correlation names for tables, these are printed after the table name. When queries create and use worktables, the “FROM TABLE” prints the name of the worktable.

When your query joins one or more tables, the order of “FROM TABLE” messages in the output shows you the order in which the query optimizer joins the tables. This order is often different from the order in which the tables are listed in the `from` clause or the `where` clause of the query. The query optimizer examines all the join orders for the tables involved and picks the join order that requires the least amount of work. This query displays the join order in a three-table join:

```
select a.au_id, au_fname, au_lname
       from titles t, titleauthor ta, authors a
where a.au_id = ta.au_id
       and ta.title_id = t.title_id
       and au_lname = "Bloom"
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is SELECT.

FROM TABLE
authors
a

Nested iteration.

Index : au_lname_ix

Ascending scan.

Positioning by key.

Keys are:

au_lname

Using I/O Size 2 Kbytes.

With LRU Buffer Replacement Strategy.

FROM TABLE
titleauthor
ta

Nested iteration.

Index : auid_titleid_ix

Ascending scan.

Positioning by key.

Index contains all needed columns. Base

table will not be read.

Keys are:

au_id

Using I/O Size 2 Kbytes.

With LRU Buffer Replacement Strategy.

FROM TABLE

```

      titles
      t
      Nested iteration.
      Using Clustered Index.
      Index : title_id_ix
      Ascending scan.
      Positioning by key.
      Keys are:
        title_id
      Using I/O Size 2 Kbytes.
      With LRU Buffer Replacement Strategy.

```

The sequence of tables in this output shows the order chosen by the Adaptive Server query optimizer, which is not the order in which they were listed in the *from* clause or *where* clause:

- First, the qualifying rows from the *authors* table are located (using the search clause on *au_lname*).
- Then, those rows are joined with the *titleauthor* table (using the join clause on the *au_id* columns).
- Finally, the *titles* table is joined with the *titleauthor* table to retrieve the desired columns (using the join clause on the *title_id* columns).

FROM TABLE and Referential Integrity

When you insert or update rows in a table that has a referential integrity constraint, the *showplan* output includes “FROM TABLE” and other messages indicating the method used to access the referenced table. If auxiliary scan descriptors were needed, the output includes an “Auxiliary Scan Descriptors” message.

This *salesdetail* table definition includes a referential integrity check on the *title_id* column:

```

create table salesdetail (
    stor_id          char(4),
    ord_num          varchar(20),
    title_id         tid
        references titles(title_id),
    qty              smallint,
    discount         float )

```

An insert to *salesdetail*, or an update on the *title_id* column, requires a lookup in the *titles* table:

```

insert salesdetail values ("S245", "X23A5", "T10",
15, 40.25)

```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is INSERT.
The update mode is direct.

FROM TABLE
titles

Using Clustered Index.

Index : title_id_ix

Ascending scan.

Positioning by key.

Keys are:

title_id

Using I/O Size 2 Kbytes.

With LRU Buffer Replacement Strategy.

TO TABLE
salesdetail

The clustered index on *title_id* provides the best access method for looking up the referenced value.

TO TABLE Message

TO TABLE
tablename

When a command such as **insert**, **delete**, **update**, or **select into** modifies or attempts to modify one or more rows of a table, the "TO TABLE" message displays the name of the target table. For operations that require an intermediate step to insert rows into a worktable, "TO TABLE" indicates that the results are going to the "Worktable" table rather than to a user table. The following examples illustrate the use of the "TO TABLE" statement:

```
insert sales
values ("8042", "QA973", "12/7/95")
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is INSERT.
The update mode is direct.

TO TABLE
sales

```

update publishers
set city = "Los Angeles"
where pub_id = "1389"

QUERY PLAN FOR STATEMENT 1 (at line 1).

```

```

STEP 1
The type of query is UPDATE.
The update mode is direct.

FROM TABLE
  publishers
Nested iteration.
  Index : pub_id_ix
Ascending scan.
Positioning by key.
Keys are:
  pub_id
Using I/O Size 2 Kbytes.
With LRU Buffer Replacement Strategy.
TO TABLE
  publishers

```

The second query indicates that the *publishers* table is used as both the “FROM TABLE” and the “TO TABLE”. In the case of `update` operations, the optimizer needs to read the table that contains the row(s) to be updated, resulting in the “FROM TABLE” statement, and then needs to modify the row(s), resulting in the “TO TABLE” statement.

Nested Iteration Message

Nested Iteration.

This message indicates one or more loops through a table to return rows. Even the simplest access to a single table is an iteration.

“Nested iteration” is the default technique used to join tables and/or return rows from a single table. For each iteration, the optimizer is using one or more sets of loops to:

1. Go through a table and retrieve a row
2. Qualify the row, based on the search criteria given in the `where` clause
3. Return the row to the front end or insert it into another table (for an `insert...select` or `select into` statement)

4. Loop again to get the next row, until all rows have been retrieved, based on the scope of the search

The method by which the query accesses the rows (such as using an available index) is discussed in “showplan Messages Describing Access Methods, Caching, and I/O Cost” on page 9-23.

The only exception to “Nested iteration” is the “EXISTS TABLE: nested iteration”. See “Existence Join Message” on page 9-58 for more information.

Update Mode Messages

Adaptive Server uses different modes to perform update operations such as insert, delete, update, and select into. These methods are called **direct update mode** and **deferred update mode**.

Direct Update Mode

The update mode is direct.

Whenever possible, Adaptive Server uses direct update mode, since it is faster and generates fewer log records than deferred update mode.

The direct update mode operates as follows:

1. Pages are read into the data cache.
2. The changes are recorded in the transaction log.
3. The change is made to the data page.
4. The transaction log page is flushed to disk when the transaction commits.

Adaptive Server uses direct update mode in the following circumstances:

- For all insert commands, unless the table into which the rows are being inserted is being read from in the same command (for example, an insert...select to a table, from the same table).
- When you create a table and populate it with a select into command, Adaptive Server uses direct update mode to insert the new rows.
- Delete operations are performed in direct update mode, unless the delete statement includes a join or columns used for referential integrity.

- Adaptive Server processes **update** commands in direct update mode or deferred update mode, depending on information that is available only at run time. For more information on the different types of direct updates, see “Update Operations” on page 8-34.

For example, Adaptive Server uses direct update mode for the following delete command:

```
delete
from authors
where au_lname = "Willis"
and au_fname = "Max"
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

```
STEP 1
  The type of query is DELETE.
  The update mode is direct.

FROM TABLE
  authors
Nested iteration.
Index : au_names
Ascending scan.
Positioning by key.
Keys are:
  au_lname
  au_fname
Using I/O Size 2 Kbytes.
With LRU Buffer Replacement Strategy.
TO TABLE
  authors
```

Deferred Mode

The update mode is deferred.

In deferred mode, processing takes place in these steps:

1. For each qualifying data row, Adaptive Server writes transaction log records for one deferred delete and one deferred insert.
2. Adaptive Server scans the transaction log to process the deferred inserts, changing the data pages and any affected index pages.

Deferred mode is used:

- For insert...select operations from a table into the same table
- For some updates (see “Update Operations” on page 8-34)

- For delete statements that include a join or columns used for referential integrity

Consider the following insert...select operation, where *mytable* is a heap without a clustered index or a unique nonclustered index:

```
insert mytable
  select title, price * 2
  from mytable
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is INSERT.
The update mode is deferred.

FROM TABLE
 mytable

Nested iteration.

Table Scan.

Ascending scan.

Positioning at start of table.

Using I/O Size 2 Kbytes.

With LRU Buffer Replacement Strategy.

TO TABLE
 mytable

This command copies every row in the table and appends the rows to the end of the table. The query processor needs to differentiate between the rows that are currently in the table (prior to the insert command) and the rows being inserted so that it does not get into a continuous loop of selecting a row, inserting it at the end of the table, selecting the row that it just inserted, and reinserting it. The query processor solves this problem by performing the operation in two steps:

1. It scans the existing table and writes insert records into the transaction log for each row that it finds.
2. When all the "old" rows have been read, it scans the log and performs the insert operations.

Deferred Index and Deferred Varcol Messages

The update mode is deferred_varcol.

The update mode is deferred_index.

These **showplan** messages indicate that Adaptive Server may process an **update** command as a deferred index update.

Adaptive Server uses `deferred_varcol` mode when updating one or more variable-length columns. This update may be done in deferred or direct mode, depending on information that is available only at run time.

Adaptive Server uses `deferred_index` mode when the index used to find to row is unique or may change as part of the update. In this mode, Adaptive Server deletes the index entries in direct mode but inserts them in deferred mode.

For more information about how deferred index updates work, see “Deferred Index Inserts” on page 8-39.

showplan Messages for Query Clauses

Use of certain Transact-SQL clauses, functions, and keywords is reflected in **showplan** output. These include **group by**, aggregates, **distinct**, **order by**, and **select into** clauses.

Table 9-2: **showplan** messages for various clauses

Message	Explanation	See
GROUP BY	The query contains a group by statement.	Page 9-13
The type of query is SELECT (into WorktableN)	The step creates a worktable to hold intermediate results.	Page 9-13
Evaluate Grouped <i>type</i> AGGREGATE or Evaluate Ungrouped <i>type</i> AGGREGATE	The query contains an aggregate. “Grouped” indicates that there is a grouping column for the aggregate (vector aggregate); “Ungrouped” indicates there is no grouping column. The variable indicates the type of aggregate.	Page 9-14 Page 9-17
Evaluate Grouped ASSIGNMENT OPERATOR Evaluate Ungrouped ASSIGNMENT OPERATOR	Query includes compute (ungrouped) or compute by (grouped).	Page 9-16
WorktableN created for DISTINCT.	The query contains a distinct keyword in the select list that requires a sort to eliminate duplicates.	Page 9-19

Table 9-2: showplan messages for various clauses (continued)

Message	Explanation	See
WorktableN created for ORDER BY.	The query contains an order by clause that requires ordering rows.	Page 9-21
This step involves sorting.	The query includes an order by or distinct clause, and results must be sorted.	Page 9-22
Using GETSORTED.	The query created a worktable and sorted it. GETSORTED is a technique used to return the rows.	Page 9-22
The sort for WorktableN is done in serial. The sort for WorktableN is done in parallel.	Indicates how the sort for a worktable is performed.	Page 9-22

GROUP BY Message

GROUP BY

This statement appears in the **showplan** output for any query that contains a **group by** clause. Queries that contain a **group by** clause are always executed in at least two steps:

- One step selects the qualifying rows into a worktable and groups them.
- Another step returns the rows from the worktable.

Selecting into a Worktable

The type of query is `SELECT (into WorktableN)`.

Queries using a **group by** clause first put qualifying results into a worktable. The data is grouped as the table is generated. A second step returns the grouped rows.

The following example returns a list of all cities and indicates the number of authors that live in each city. The query plan shows the two steps: the first step selects the rows into a worktable, and the second step retrieves the grouped rows from the worktable:

```
select city, total_authors = count(*)
  from authors
  group by city
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1
  The type of query is SELECT (into
  Worktable1).
  GROUP BY
  Evaluate Grouped COUNT AGGREGATE.

  FROM TABLE
    authors
  Nested iteration.
  Table Scan.
  Ascending scan.
  Positioning at start of table.
  Using I/O Size 2 Kbytes.
  With LRU Buffer Replacement Strategy.
  TO TABLE
    Worktable1.

STEP 2
  The type of query is SELECT.

  FROM TABLE
    Worktable1.
  Nested iteration.
  Table Scan.
  Ascending scan.
  Positioning at start of table.
  Using I/O Size 2 Kbytes.
  With MRU Buffer Replacement Strategy.
```

Grouped Aggregate Message

Evaluate Grouped *type* AGGREGATE

This message is printed by queries that contain aggregates and **group by** or **compute by**.

The variable indicates the type of aggregate—COUNT, SUM OR AVERAGE, MINIMUM, or MAXIMUM.

avg reports both COUNT and SUM OR AVERAGE; **sum** reports SUM OR AVERAGE. Two additional types of aggregates (ONCE and ANY) are used internally by Adaptive Server while processing subqueries. See “Internal Subquery Aggregates” on page 9-53.

Grouped Aggregates and *group by*

When an aggregate function is combined with **group by**, the result is called a grouped aggregate, or **vector aggregate**. The query results have one row for each value of the grouping column or columns.

The following example illustrates a grouped aggregate:

```
select type, avg(advance)
from titles
group by type
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is SELECT (into Worktable1).
GROUP BY
Evaluate Grouped COUNT AGGREGATE.
Evaluate Grouped SUM OR AVERAGE AGGREGATE.

FROM TABLE
titles
Nested iteration.
Table Scan.
Ascending scan.
Positioning at start of table.
Using I/O Size 2 Kbytes.
With LRU Buffer Replacement Strategy.
TO TABLE
Worktable1.

STEP 2

The type of query is SELECT.

FROM TABLE
Worktable1.
Nested iteration.
Table Scan.
Ascending scan.
Positioning at start of table.
Using I/O Size 2 Kbytes.
With MRU Buffer Replacement Strategy.

In the first step, the worktable is created, and the aggregates are computed. The second step selects the results from the worktable.

compute by Message

Evaluate Grouped ASSIGNMENT OPERATOR

Queries using **compute by** display the same aggregate messages as **group by**, as well as the "Evaluate Grouped ASSIGNMENT OPERATOR" message. The values are placed in a worktable in one step, and the computation of the aggregates is performed in a second step. This query uses *type* and *advance*, like the **group by** query example above:

```
select type, advance from titles
having title like "Compu%"
order by type
compute avg(advance) by type
```

In the **showplan** output, the computation of the aggregates takes place in Step 2:

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is INSERT.
The update mode is direct.
Worktable1 created for ORDER BY.

FROM TABLE
titles
Nested iteration.
Index : title_ix
Ascending scan.
Positioning by key.
Keys are:
title
Using I/O Size 2 Kbytes.
With LRU Buffer Replacement Strategy.
TO TABLE
Worktable1.

STEP 2

The type of query is SELECT.
Evaluate Grouped SUM OR AVERAGE AGGREGATE.
Evaluate Grouped COUNT AGGREGATE.
Evaluate Grouped ASSIGNMENT OPERATOR.

This step involves sorting.

```
FROM TABLE
    Worktable1.
Using GETSORTED
Table Scan.
Ascending scan.
Positioning at start of table.
Using I/O Size 2 Kbytes.
With MRU Buffer Replacement Strategy.
```

Ungrouped Aggregate Message

Evaluate Ungrouped *type* AGGREGATE.

This message is reported by:

- Queries that use aggregate functions, but do not use **group by**
- Queries that use **compute**

See “Grouped Aggregate Message” on page 9-14 for an explanation of *type*.

Ungrouped Aggregates

When an aggregate function is used in a select statement that does not include a **group by** clause, it produces a single value. The query can operate on all rows in a table or on a subset of the rows defined by a **where** clause. When an aggregate function produces a single value, the function is called a **scalar aggregate**, or an ungrouped aggregate. Here is **showplan** output for an ungrouped aggregate:

```
select avg(advance)
from titles
where type = "business"
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is SELECT.

Evaluate Ungrouped COUNT AGGREGATE.

Evaluate Ungrouped SUM OR AVERAGE AGGREGATE.

```
FROM TABLE
    titles
Nested iteration.
```

```

Index : type_ix
Ascending scan.
Positioning by key.
Keys are:
    type
Using I/O Size 2 Kbytes.
With LRU Buffer Replacement Strategy.

```

STEP 2

The type of query is SELECT.

Notice that **showplan** considers this a two-step query, which is similar to the **showplan** from the **group by** query shown earlier. Since the scalar aggregate returns a single value, Adaptive Server uses an internal variable to compute the result of the aggregate function, as the qualifying rows from the table are evaluated. After all rows from the table have been evaluated (Step 1), the final value from the variable is selected (Step 2) to return the scalar aggregate result.

compute Messages

```
Evaluate Ungrouped ASSIGNMENT OPERATOR
```

When a query includes **compute** to compile a scalar aggregate, **showplan** prints the “Evaluate Ungrouped ASSIGNMENT OPERATOR” message. This query computes an average for the entire result set:

```

select type, advance from titles
having title like "Compu%"
order by type
compute avg(advance)

```

The **showplan** output shows that the computation of the aggregate values takes place in the second step:

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
```

STEP 1

```

The type of query is INSERT.
The update mode is direct.
Worktable1 created for ORDER BY.

```

```

FROM TABLE
    titles
Nested iteration.
Index : titles_ix
Ascending scan.
Positioning by key.

```

```

Keys are:
  title
Using I/O Size 2 Kbytes.
With LRU Buffer Replacement Strategy.
TO TABLE
  Worktable1.

STEP 2
The type of query is SELECT.
Evaluate Ungrouped SUM OR AVERAGE AGGREGATE.
Evaluate Ungrouped COUNT AGGREGATE.
Evaluate Ungrouped ASSIGNMENT OPERATOR.
This step involves sorting.

FROM TABLE
  Worktable1.
Using GETSORTED
Table Scan.
Ascending scan.
Positioning at start of table.
Using I/O Size 2 Kbytes.
With MRU Buffer Replacement Strategy.

```

Messages for *order by* and *distinct*

Some queries that include *distinct* use a sort step to locate the duplicate values in the result set. *distinct* queries and *order by* queries can avoid the sorting step when the indexes used to locate rows support the *order by* or *distinct* clause.

For those cases where the sort must be performed, the *distinct* keyword in a select list and the *order by* clause share some *showplan* messages:

- Each generates a worktable message
- The message “This step involves sorting.”
- The message “Using GETSORTED”

Worktable Message for *distinct*

```
WorktableN created for DISTINCT.
```

A query that includes the *distinct* keyword excludes all duplicate rows from the results so that only unique rows are returned. When there is no useful index, Adaptive Server performs these steps to process queries that include *distinct*:

1. It creates a worktable to store all of the results of the query, including duplicates.
2. It sorts the rows in the worktable, discarding the duplicate rows, and then returns the rows.

The “WorktableN created for DISTINCT” message appears as part of “Step 1” in showplan output. “Step 2” for distinct queries includes the messages “This step involves sorting” and “Using GETSORTED”. See “Sorting Messages” on page 9-22.

```
select distinct city
from authors
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
```

```
STEP 1
```

```
The type of query is INSERT.
```

```
The update mode is direct.
```

```
Worktable1 created for DISTINCT.
```

```
FROM TABLE
```

```
authors
```

```
Nested iteration.
```

```
Table Scan.
```

```
Ascending scan.
```

```
Positioning at start of table.
```

```
Using I/O Size 2 Kbytes.
```

```
With LRU Buffer Replacement Strategy.
```

```
TO TABLE
```

```
Worktable1.
```

```
STEP 2
```

```
The type of query is SELECT.
```

```
This step involves sorting.
```

```
FROM TABLE
```

```
Worktable1.
```

```
Using GETSORTED
```

```
Table Scan.
```

```
Ascending scan.
```

```
Positioning at start of table.
```

```
Using I/O Size 2 Kbytes.
```

```
With MRU Buffer Replacement Strategy.
```


Worktable Message for *order by*

WorktableN created for ORDER BY.

Queries that include an **order by** clause often require the use of a temporary worktable. When the optimizer cannot use an index to order the result rows, it creates a worktable to sort the result rows before returning them. This example shows an **order by** clause that creates a worktable because there is no index on the *city* column:

```
select *  
from authors  
order by city
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is INSERT.

The update mode is direct.

Worktable1 created for ORDER BY.

FROM TABLE

authors

Nested iteration.

Table Scan.

Ascending scan.

Positioning at start of table.

Using I/O Size 2 Kbytes.

With LRU Buffer Replacement Strategy.

TO TABLE

Worktable1.

STEP 2

The type of query is SELECT.

This step involves sorting.

FROM TABLE

Worktable1.

Using GETSORTED

Table Scan.

Ascending scan.

Positioning at start of table.

Using I/O Size 2 Kbytes.

With MRU Buffer Replacement Strategy.

order by Queries and Indexes

Certain queries using *order by* do not require a sorting step, depending on the type of index used to access the data. See “Optimizing Indexes for Queries That Perform Sorts” on page 7-25 for more information.

Sorting Messages

These messages report on sorts.

This Step Involves Sorting Message

`This step involves sorting.`

This `showplan` message indicates that the query must sort the intermediate results before returning them to the user. Queries that use `distinct` or that have an `order by` clause not supported by an index require an intermediate sort. The results are put into a worktable, and the worktable is then sorted. For examples of this message, see “Worktable Message for `distinct`” on page 9-19 and “Worktable Message for `order by`” on page 9-21.

GETSORTED Message

`Using GETSORTED`

This statement indicates one of the ways that Adaptive Server returns result rows from a table. In the case of “Using `GETSORTED`,” the rows are returned in sorted order. However, not all queries that return rows in sorted order include this step. For example, `order by` queries whose rows are retrieved using an index with a matching sort sequence do not require “`GETSORTED`.”

The “Using `GETSORTED`” method is used when Adaptive Server must first create a temporary worktable to sort the result rows and then return them in the proper sorted order. The examples for `distinct` on Page 9-19 and for `order by` on Page 9-21 show the “Using `GETSORTED`” message.

Serial or Parallel Sort Message

`The sort for WorktableN is done in Serial.`

`The sort for WorktableN is done in Parallel.`

These messages indicate whether a serial or parallel sort was performed for a worktable. They are printed by the sort manager after it determines whether a given sort should be performed in parallel or in serial. If set `noexec` is in effect, the worktable is not created, so the sort manager is not invoked, and no message is displayed.

***showplan* Messages Describing Access Methods, Caching, and I/O Cost**

`showplan` output provides information about access methods and caching strategies.

Table 9-3: `showplan` messages describing access methods

Message	Explanation	See
Auxiliary scan descriptors required: <i>N</i>	<i>N</i> indicates the number of scan descriptors needed to run the query.	Page 9-24
Table Scan.	Indicates that the query performs a table scan.	Page 9-26
Using Clustered Index.	Query uses the clustered index on the table.	Page 9-27
Index : <i>index_name</i>	Query uses an index on the table; the variable shows the index name.	Page 9-27
Ascending scan.	Indicates an ascending table or index scan.	Page 9-28
Descending scan.	Indicates a descending table or index scan.	Page 9-28
Positioning at start of table. Positioning by Row Identifier (RID). Positioning by key. Positioning at index start. Positioning at index end.	These messages indicate how scans are taking place.	Page 9-28
Scanning only up to the first qualifying row. Scanning only the last page of the table.	These messages indicate <code>min</code> and <code>max</code> optimization, respectively.	Page 9-29
Index contains all needed columns. Base table will not be read.	Indicates that the nonclustered index covers the query.	Page 9-30

Table 9-3: showplan messages describing access methods (continued)

Message	Explanation	See
Keys are:	Included when the positioning message indicates "Positioning by key." The next line(s) shows the index key(s) used.	Page 9-31
Using N Matching Index Scans	Indicates that a query with in or or is performing multiple index scans, one for each or condition or in (<i>values list</i>) item.	Page 9-32
Using Dynamic Index.	Reported during some queries using or clauses or in (<i>values list</i>).	Page 9-33
WorktableN created for REFORMATTING.	Indicates that an inner table of a join has no useful indexes, and that Adaptive Server has determined that it is cheaper to build a worktable and an index on the worktable than to perform repeated table scans.	Page 9-35
Log Scan.	Query fired a trigger that uses <i>inserted</i> or <i>deleted</i> tables.	Page 9-37
Using I/O size N Kbytes.	Variable indicates the I/O size for disk reads and writes.	Page 9-37
With LRU/MRU buffer replacement strategy.	Reports the caching strategy for the table.	Page 9-38
Total estimated I/O cost for statement N (at line N): X.	"Statement N" is the statement number within the batch, "line N" is the line number within the batch, and X is the cost estimate.	Page 9-38

Auxiliary Scan Descriptors Message

Auxiliary scan descriptors required: N

When a query involving referential integrity requires auxiliary scan descriptors, this `showplan` message indicates the number of descriptors needed. If the query does not exceed the number of pre-allocated scan descriptors allotted for the session, the "Auxiliary scan descriptors required" message is not included in the `showplan` output.

The following example shows output for a delete from a referenced table, *employees*, that is referenced by 30 foreign tables:

```
delete employees
where empl_id = "222-09-3482"
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
```

```
Auxiliary scan descriptors required: 6
```

```
STEP 1
```

```
The type of query is DELETE.
The update mode is deferred.
```

```
FROM TABLE
    employees
Nested iteration.
Using Clustered Index.
Index : employees_empl_i_8000058811
Ascending scan.
Positioning by key.
Keys are:
    empl_id
Using I/O Size 2 Kbytes.
With LRU Buffer Replacement Strategy.
```

```
FROM TABLE
    payroll
Index : payroll_empl_id_ncix
Ascending scan.
Positioning by key.
Index contains all needed columns. Base
table will not be read.
Keys are:
    empl_id
Using I/O Size 2 Kbytes.
With LRU Buffer Replacement Strategy.
```

```
...
```

```
FROM TABLE
    benefits
Index : benefits_empl_id_ncix
```

```
Ascending scan.  
Positioning by key.  
Index contains all needed columns. Base  
table will not be read.  
Keys are:  
  empl_id  
Using I/O Size 2 Kbytes.  
With LRU Buffer Replacement Strategy.  
TO TABLE  
  employees
```

Table Scan Message

Table Scan.

This message indicates that the query performs a table scan.

When a table scan is performed, the execution begins with the first row in the table. Each row is retrieved, compared to the conditions in the *where* clause, and returned to the user, if it meets the query criteria. Every row in the table must be looked at, so for very large tables, a table scan can be very costly in terms of disk I/O. If a table has one or more indexes on it, the query optimizer may still choose to do a table scan instead of using one of the available indexes, if the indexes are too costly or are not useful for the given query. The following query shows a typical table scan:

```
select au_lname, au_fname  
from authors
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

```
STEP 1  
  The type of query is SELECT.  
  
  FROM TABLE  
    authors  
  Nested iteration.  
  Table Scan.  
  Ascending scan.  
  Positioning at start of table.  
  Using I/O Size 2 Kbytes.  
  With LRU Buffer Replacement Strategy.
```

Clustered Index Message

Using Clustered Index.

This **showplan** message indicates that the query optimizer chose to use the clustered index on a table to retrieve the rows. The following query shows the clustered index being used to retrieve the rows from the table:

```
select title_id, title
from titles
where title_id like "T9%"
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is SELECT.

FROM TABLE
titles

Nested iteration.

Using Clustered Index.

Index : title_id_ix

Ascending scan.

Positioning by key.

Keys are:

title_id

Using I/O Size 2 Kbytes.

With LRU Buffer Replacement Strategy.

Index Name Message

Index : *indexname*

This message indicates that the optimizer is using an index to retrieve the rows. The message includes the index name. If the line above this message in the output is "Using Clustered Index," the index is clustered; otherwise, the index is nonclustered.

The keys used to position the search are reported in the "Keys are..." message. See "Keys are: *keys_list*" on page 9-31.

This query illustrates the use of a nonclustered index to find and return rows:

```
select au_id, au_fname, au_lname
from authors
where au_fname = "Susan"
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1
  The type of query is SELECT.

  FROM TABLE
    authors
  Nested iteration.
  Index : au_name_ix
  Ascending scan.
  Positioning by key.
  Keys are:
    au_fname
  Using I/O Size 2 Kbytes.
  With LRU Buffer Replacement Strategy.
```

Scan Direction Messages

Ascending scan.

Descending scan.

These messages indicate the direction of the table or index scan. Descending scans may be used when the `order by` clause contains the `desc` qualifier and the configuration parameter `allow backward scans` is set to 1. All other accesses use ascending scans. When a query includes `order by...desc`, and `allow backward scans` is set to 0, the query creates a worktable and performs a sort. For more information on indexes and sort operations, see “Optimizing Indexes for Queries That Perform Sorts” on page 7-25.

Positioning Messages

Positioning at start of table.

Positioning by Row Identifier (RID).

Positioning by key.

Positioning at index start.

Positioning at index end.

These messages describe how access to a table or to the leaf level of a nonclustered index takes place. The choices are:

- Positioning at start of table.

This message indicates a table scan, starting at the first row of the table.

- Positioning by Row IDentifier (RID).

This message is printed after the OR strategy has created a dynamic index of row IDs. See “Dynamic Index Message” on page 9-33 for more information about how row IDs are used.

- Positioning by key.

This messages indicates that the index is used to find the qualifying row or the first qualifying row. It is printed for:

- Direct access to individual rows in point queries
- Range queries that perform matching scans of the leaf level of a nonclustered index
- Range queries that scan the data pages when there is a clustered index
- Indexed accesses to inner tables in joins

- Positioning at index start.

Positioning at index end.

These messages indicate a nonmatching, nonclustered index scan, which is used when the index covers the query. Matching scans are positioned by key.

Ascending scans are positioned at the start of the index; descending scans are positioned at the end of the index. The only time descending scans are used is when the order by clause includes the desc qualifier.

Scanning Messages

Scanning only the last page of the table.

This message indicates that a query containing an ungrouped (scalar) max aggregate needs to access only the last page of the table. In order to use this special optimization, the aggregate column needs to be the leading column in the index. See “Optimizing Aggregates” on page 8-26 for more information.

Scanning only up to the first qualifying row.

This message appears only for queries that use an ungrouped (scalar) min aggregate. The aggregated column needs to be the leading column in the index. See “Optimizing Aggregates” on page 8-26 for more information.

Index Covering Message

Index contains all needed columns. Base table will not be read.

This message indicates that the nonclustered index covers the query. It is printed both for matching index accesses and for non-matching scans. The difference in `showplan` output for the two types of queries can be seen from two other parts of the `showplan` output for the query:

- A matching scan reports “Positioning by key.” A nonmatching scan reports “Positioning at index start,” since a nonmatching scan must read the entire leaf level of the nonclustered index.
- If the optimizer uses a matching scan, the “Keys are...” message reports the keys used to position the search. This information is not included for a nonmatching scan, since the keys are not used for positioning, but only for selecting the rows to return.

The next query shows output for a matching scan, using a composite, nonclustered index on `au_lname`, `au_fname`, `au_id`:

```
select au_fname, au_lname, au_id
from authors
where au_lname = "Williams"

QUERY PLAN FOR STATEMENT 1 (at line 1).
```

STEP 1

The type of query is SELECT.

FROM TABLE
authors

Nested iteration.

Index : au_names_id

Ascending scan.

Positioning by key.

Index contains all needed columns. Base table will not be read.

Keys are:

au_lname

Using I/O Size 2 Kbytes.

With LRU Buffer Replacement Strategy.

The index is used to find the first occurrence of “Williams” on the nonclustered leaf page. The query scans forward, looking for more occurrences of “Williams” and returning any it finds. Once a value greater than “Williams” is found, the query has found all the matching values, and the query stops. All the values needed in the where clauses and select list are included in this index, so no access to the table is required.

With the same composite index on *au_lname*, *au_fname*, *au_id*, this query performs a nonmatching scan, since the leading column of the index is not included in the where clause:

```
select au_fname, au_lname, au_id
from authors
where au_id = "A93278"
```

Note that the showplan output does not contain a “Keys are...” message, and the positioning message is “Positioning at index start.”

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
```

```
STEP 1
  The type of query is SELECT.

  FROM TABLE
    authors
  Nested iteration.
  Index : au_names_id
  Ascending scan.
  Positioning at index start.
  Index contains all needed columns. Base
table will not be read.
  Using I/O Size 2 Kbytes.
  With LRU Buffer Replacement Strategy.
```

This query must scan the entire leaf level of the nonclustered index, since the rows are not ordered and there is no way for Adaptive Server to predict the uniqueness of a particular column in a composite index.

Keys Message

```
Keys are:
  keys_list
```

This message is followed by the key(s) used whenever Adaptive Server uses a clustered or a matching, nonclustered index scan to

locate rows. For composite indexes, all keys in the **where** clauses are listed. Examples are included under those messages.

Matching Index Scans Message

Using *N* Matching Index Scans.

This **showplan** message indicates that a query using **or** clauses or an **in** (*values list*) is using multiple index scans to return the rows directly without using a dynamic index. See “Dynamic Index Message” on page 9-33 for information on this strategy.

Multiple matching scans can be used only when there is no possibility that the **or** clauses or **in** list items will match duplicate rows—that is, when there is no need to build the worktable and perform the sort to remove the duplicates. For example, this query can use the multiple matching scans strategy because the two scans will never return duplicate rows:

```
select * from titles
where type = "business" or type = "art"
```

This query creates a dynamic index in order to avoid returning duplicate rows:

```
select * from titles
where type = "business" or title = "T752340"
```

In some cases, different indexes may be used for some of the scans, so the messages that describe the type of index, index positioning, and keys used are printed for each scan.

The following example uses multiple matching index scans to return rows:

```
select title
from titles
where title_id in ("T18168","T55370")
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

```
STEP 1
The type of query is SELECT.

FROM TABLE
    titles
Nested iteration.
Using 2 Matching Index Scans
Index : title_id_ix
```

```
Ascending scan.  
Positioning by key.  
Keys are:  
    title_id  
Index : title_id_ix  
Ascending scan.  
Positioning by key.  
Keys are:  
    title_id  
Using I/O Size 2 Kbytes.  
With LRU Buffer Replacement Strategy.
```

Dynamic Index Message

Using Dynamic Index.

The term **dynamic index** refers to a special table of row IDs used to process queries that use `or` clauses. When a query contains `or` clauses or an `in (values list)` clause, Adaptive Server will do one of the following, depending on query costing:

- Scan the table once, finding all rows that match each of the conditions. You will see a “Table Scan” message, but a dynamic index will not be used.
- Use one or more indexes and access the table once for each `or` clause or item in the `in` list. You will see a “Positioning by key” message and the “Using Dynamic Index” message. This technique is called the **OR strategy**. For a full explanation, see “Optimization of `or` clauses and `in (values_list)`” on page 8-23.

When the OR strategy is used, Adaptive Server builds a list of all the row IDs that match the query, sorts the list to remove duplicates, and uses the list to retrieve the rows from the table.

Adaptive Server does not use the OR strategy for all queries that contain `or` clauses. The following conditions must be met:

- All columns in the `or` clause must belong to the same table.
- If any portion of the `or` clause requires a table scan (due to lack of an appropriate index or poor selectivity of a given index), then a table scan is used for the entire query.

If the query contains **or** clauses on different columns in the same table, and each of those columns has a useful index, Adaptive Server can use different indexes for each clause.

The OR strategy cannot be used for cursors.

The showplan output below includes three “FROM TABLE” sections:

- The first two “FROM TABLE” blocks in the output show the two index accesses, one for “Bill” and one for “William”.
- The final “FROM TABLE” block shows the “Using Dynamic Index” output with its companion positioning message, “Positioning by Row Identifier (RID).” This is the step where the dynamic index is used to locate the table rows to be returned.

```
select au_id, au_fname, au_lname
from authors
where au_fname = "Bill"
      or au_fname = "William"
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is SELECT.

FROM TABLE
authors

Nested iteration.

Index : au_fname_ix

Ascending scan.

Positioning by key.

Keys are:

au_fname

Using I/O Size 2 Kbytes.

With LRU Buffer Replacement Strategy.

FROM TABLE
authors

Nested iteration.

Index : au_fname_ix

Ascending scan.

Positioning by key.

Keys are:

au_fname

Using I/O Size 2 Kbytes.

```
With LRU Buffer Replacement Strategy.  
  
FROM TABLE  
    authors  
Nested iteration.  
Using Dynamic Index.  
Ascending scan.  
Positioning by Row Identifier (RID).  
Using I/O Size 2 Kbytes.  
With LRU Buffer Replacement Strategy.
```

Reformatting Message

```
WorktableN Created for REFORMATTING.
```

When joining two or more tables, Adaptive Server may choose to use a **reformatting strategy** to join the tables when the tables are large and the tables in the join do not have a useful index.

The reformatting strategy:

- Inserts the needed columns from qualifying rows of the smaller of the two tables into a worktable.
- Creates a clustered index on the join column(s) of the worktable. The index is built using the keys that join the worktable to the outer table in the query.
- Uses the clustered index in the join to retrieve the qualifying rows from the table.

See “Saving I/O Using the Reformatting Strategy” on page 8-17 for more information on reformatting.

► **Note**

If your queries frequently employ the reformatting strategy, examine the tables involved in the query. Unless there are other overriding factors, you may want to create an index on the join columns of the table.

The following example illustrates the reformatting strategy. It performs a three-way join on the *titles*, *titleauthor*, and *titles* tables. There are no indexes on the join columns in the tables (*au_id* and *title_id*), which forces Adaptive Server to use the reformatting strategy on two of the tables:

```
select au_lname, title
from authors a, titleauthor ta, titles t
where a.au_id = ta.au_id
and t.title_id = ta.title_id
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

```
The type of query is INSERT.
The update mode is direct.
Worktable1 created for REFORMATTING.
FROM TABLE
    titleauthor
    ta
Nested iteration.
Table Scan.
Ascending scan.
Positioning at start of table.
Using I/O Size 2 Kbytes.
With LRU Buffer Replacement Strategy.
TO TABLE
    Worktable1.
```

STEP 2

```
The type of query is INSERT.
The update mode is direct.
Worktable2 created for REFORMATTING.

FROM TABLE
    authors
    a
Nested iteration.
Table Scan.
Ascending scan.
Positioning at start of table.
Using I/O Size 2 Kbytes.
With LRU Buffer Replacement Strategy.
TO TABLE
    Worktable2.
```

STEP 3

```
The type of query is SELECT.

FROM TABLE
    titles
    t
Nested iteration.
```



```
Table Scan.  
Ascending scan.  
Positioning at start of table.  
Using I/O Size 2 Kbytes.  
With LRU Buffer Replacement Strategy.
```

```
FROM TABLE  
    Worktable1.  
Nested iteration.  
Using Clustered Index.  
Ascending scan.  
Positioning by key.  
Using I/O Size 2 Kbytes.  
With LRU Buffer Replacement Strategy.
```

```
FROM TABLE  
    Worktable2.  
Nested iteration.  
Using Clustered Index.  
Ascending scan.  
Positioning by key.  
Using I/O Size 2 Kbytes.  
With LRU Buffer Replacement Strategy.
```

Trigger Log Scan Message

Log Scan.

When an insert, update, or delete statement causes a trigger to fire, and the trigger includes access to the *inserted* or *deleted* tables, these tables are built by scanning the transaction log.

I/O Size Message

Using I/O size *N* Kbytes.

This message reports the I/O size used in the query. For tables and indexes, the possible sizes are 2K, 4K, 8K, and 16K. If the table, index, or database used in the query uses a data cache with large I/O sized pools, the Adaptive Server optimizer can choose to use large I/O for some types of queries.

See Chapter 16, “Memory Use and Performance,” for more information on large I/Os and the data cache.

Cache Strategy Message

With <LRU/MRU> Buffer Replacement Strategy.

Indicates the cache strategy used by the query. See “Overview of Cache Strategies” on page 3-16 for more information on cache strategies.

Total Estimated I/O Cost Message

Total estimated I/O cost for statement *N* (at line *N*): *X*.

Adaptive Server prints this message only if a System Administrator has configured Adaptive Server to enable resource limits. Adaptive Server prints this line once for each query in a batch. The message displays the optimizer’s estimate of the total cost of logical and physical I/O. The result is printed as a unitless number, that is, the value is not based on a single unit of measurement, such as seconds or milliseconds. System Administrators can use this value when setting compile-time resource limits.

The cost estimate returned by `showplan` may differ from the actual cost returned by `statistics io`. The optimizer calculates the estimated cost before the query is run, based on index statistics and query clauses. The optimizer does not know how many pages are already in the cache, so it makes the pessimistic assumption that it will need to do physical I/O. The actual cost provided by `statistics io` will be much lower than the estimated cost, if the pages are already in cache.

Used in conjunction with `set noexec on`, `showplan` allows you to get an estimate of I/O cost without performing the actual work. For information on using `set noexec on`, see “Combining `showplan` and `noexec`” on page 9-1.

The following example demonstrates `showplan` output for an Adaptive Server configured to allow resource limits:

```
select au_lname, au_fname
from authors
where city = "Oakland"
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is SELECT.

FROM TABLE

authors

Nested iteration.

Table Scan.

Ascending scan.

Positioning at start of table.

Using I/O Size 2 Kbytes.

With LRU Buffer Replacement Strategy.

Total estimated I/O cost for statement 1 (at line 1): 5463.

For more information on creating resource limits, see Chapter 9, “Understanding Query Plans,” in the *System Administration Guide*.

showplan Messages for Parallel Queries

showplan reports information about parallel execution, explicitly stating which query steps are executed in parallel.

Table 9-4: *showplan* messages for parallel queries

Message	Explanation	See
Executed in parallel by coordinating process and <i>N</i> worker processes.	Indicates that a query is run in parallel, and shows the number of worker processes used.	Page 9-40
Executed in parallel by <i>N</i> worker processes.	Indicates the number of worker processes used for a query step.	Page 9-41
Executed in parallel with a <i>N</i> -way hash scan.	Indicates the number of worker processes and the type of scan, hash-based or partition-based, for a query step.	Page 9-41
Executed in parallel with a <i>N</i> -way partition scan.		
Parallel work table merge. Parallel network buffer merge. Parallel result buffer merge.	Indicates the way in which the results of parallel scans were merged.	Page 9-42

Table 9-4: showplan messages for parallel queries (continued)

Message	Explanation	See
AN ADJUSTED QUERY PLAN WILL BE USED FOR STATEMENT <i>N</i> BECAUSE NOT ENOUGH WORKER PROCESSES ARE AVAILABLE AT THIS TIME. ADJUSTED QUERY PLAN:	Indicates that a run-time adjustment to the number of worker processes was required.	Page 9-45

Executed in Parallel Messages

The Adaptive Server optimizer uses parallel query optimization strategies only when a given query is eligible for parallel execution. If the query is processed in parallel, `showplan` uses three separate messages to report:

- The fact that some or all of the query was executed by the coordinating process and worker processes. The number of worker processes is included in this message.
- The number of worker processes for each step of the query that is executed in parallel.
- The degree of parallelism for each scan.

Note that the degree of parallelism used for a query step is not the same as the total number of worker processes used for the query. For more examples of parallel query plans, see Chapter 14, “Parallel Query Optimization.”

Coordinating Process Message

Executed in parallel by coordinating process and *N* worker processes.

For each query that is eligible for parallel execution, `showplan` reports on the coordinating process and the number of worker processes following the “Query plan” message.

The limiting factors for the number of worker processes in a statement are:

- The configured value of `number of worker processes`
- The configured value of `max parallel degree`

- The maximum number of worker processes for the steps in the statement

When Adaptive Server executes a query in parallel, the coordinating process initiates execution, connects to the specified number of worker processes, sends synchronization messages to the worker processes, and controls the merging of results from the worker processes.

Worker Processes Message

Executed in parallel by *N* worker processes.

For each step in a query that is executed in parallel, **showplan** reports the number of worker processes for the step following the “Type of query” message.

The limiting factors for the number of worker processes in a step are:

- The configured value of **number of worker processes**.
- The configured value of **max parallel degree**.
- The product of the degrees of parallelism for the scans in the step, unless the query contains **group by all**.

group by all queries require two steps that insert into a worktable. The number of worker processes for the first step is the larger of:

- the product of the scans in the first step
- the product of the scans in the second step

The worktable created for the **group by** must be partitioned with the number of partitions required for the largest degree of parallelism.

Scan Type Message

Executed in parallel with a *N*-way hash scan.

Executed in parallel with a *N*-way partition scan.

For each step in the query that accesses data in parallel, **showplan** prints the number of worker processes used for the scan, and the type of scan, either “hash” or “partition”.

The limiting factors for the number of worker processes used in a scan are:

- The number of partitions

- The values of the `max parallel degree` and `max scan parallel degree` configuration parameters
- A lower limit imposed with `set parallel_degree` or `set scan_parallel_degree`
- A lower limit specified in the `select` command

Merge Messages

A parallel merge is the process of merging results from all the worker processes participating in the parallel execution of a query. There are three types of merges:

- Parallel worktable merge
- Parallel network buffer merge
- Parallel result buffer merge

Merge Message for Worktables

Parallel work table merge.

In this type of merge, Adaptive Server merges grouped aggregate results from the worktables which are created by worker processes into one result set.

In the following example, *titles* has two partitions. The `showplan` information specific to parallel query processing appears in bold.

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

QUERY PLAN FOR STATEMENT 1 (at line 1).

Executed in parallel by coordinating process and 2 worker processes.

STEP 1

The type of query is SELECT (into Worktable1).

GROUP BY

Evaluate Grouped SUM OR AVERAGE AGGREGATE.

Executed in parallel by coordinating process and 2 worker processes.

FROM TABLE

titles

Nested iteration.

Table Scan.

Ascending scan.
 Positioning at start of table.
Executed in parallel with a 2-way partition scan.
 Using I/O Size 16 Kbytes.
 With LRU Buffer Replacement Strategy.
 TO TABLE
 Worktable1.

Parallel work table merge.

STEP 2

The type of query is SELECT.
Executed by coordinating process.

FROM TABLE
 Worktable1.
 Nested iteration.
 Table Scan.
 Ascending scan.
 Positioning at start of table.
 Using I/O Size 16 Kbytes.
 With MRU Buffer Replacement Strategy.

Merge Message for Buffer Merges

Parallel network buffer merge.

In this type of merge, Adaptive Server merges unsorted, nonaggregate results returned by the worker processes into a network buffer that is sent to the client. You can configure the size of network buffers with the default network packet size and max network packet size configuration parameters.

In the following example, *titles* has two partitions.

select title_id from titles

QUERY PLAN FOR STATEMENT 1 (at line 1).
 Executed in parallel by coordinating process and 2 worker processes.

STEP 1

The type of query is SELECT.
 Executed in parallel by coordinating process and 2 worker processes.

FROM TABLE

```

    titles
    Nested iteration.
    Table Scan.
    Ascending scan.
    Positioning at start of table.
    Executed in parallel with a 2-way partition scan.
    Using I/O Size 16 Kbytes.
    With LRU Buffer Replacement Strategy.

```

Parallel network buffer merge.

Merge Message for Result Buffers

Parallel result buffer merge.

In this type of merge, Adaptive Server merges ungrouped aggregate results or unsorted, nonaggregate variable assignment results from worker processes. Each worker process stores the aggregate in a result buffer. The result buffer merge produces a single result.

Result buffers are small data structures that range from zero-length (when the value is NULL) to the maximum length of a character string (255).

In the following example, *titles* has two partitions:

```

    select sum(total_sales)
    from titles

```

```

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 2 worker
processes.

```

STEP 1

```

    The type of query is SELECT.
    Evaluate Ungrouped SUM OR AVERAGE AGGREGATE.
    Executed in parallel by coordinating process and 2
worker processes.

```

```

FROM TABLE
    titles
    Nested iteration.
    Table Scan.
    Ascending scan.

```


Positioning at start of table.
Executed in parallel with a 2-way partition scan.
Using I/O Size 16 Kbytes.
With LRU Buffer Replacement Strategy.

Parallel result buffer merge.

STEP 2

The type of query is SELECT.
Executed by coordinating process.

AN ADJUSTED QUERY PLAN WILL BE USED FOR STATEMENT N BECAUSE NOT
ENOUGH WORKER PROCESSES ARE AVAILABLE AT THIS TIME.
ADJUSTED QUERY PLAN:

Run-Time Adjustment Message

AN ADJUSTED QUERY PLAN WILL BE USED FOR STATEMENT
N BECAUSE NOT ENOUGH WORKER PROCESSES ARE
AVAILABLE AT THIS TIME.
ADJUSTED QUERY PLAN:

If **showplan** was used with **noexec** turned off, **showplan** output displays an adjusted query plan, when fewer worker processes are available at run time than the number required by the optimized query plan. For more information on when adjusted query plans are displayed in **showplan** output, see “Using showplan” on page 14-33.

***showplan* Messages for Subqueries**

Since subqueries can contain the same clauses that regular queries contain, their **showplan** output can include many of the messages listed in earlier sections.

The **showplan** messages for subqueries, shown in Table 9-5, include special delimiters so that you can easily spot the beginning and the end of a subquery processing block, the messages that identify the type of subquery, the place in the outer query where the subquery is

executed, and messages for special types of processing that is performed only in subqueries.

Table 9-5: showplan messages for subqueries

Message	Explanation	See
Run subquery <i>N</i> (at nesting level <i>N</i>).	This message appears at the point in the query where the subquery actually runs. Subqueries are numbered in order for each side of a union.	Page 9-52
NESTING LEVEL <i>N</i> SUBQUERIES FOR STATEMENT <i>N</i> .	Shows the nesting level of the subquery.	Page 9-52
QUERY PLAN FOR SUBQUERY <i>N</i> (at line <i>N</i>).	These lines bracket showplan output for each subquery in a statement. Variables show the subquery number, the nesting level, and the input line.	Page 9-52
END OF QUERY PLAN FOR SUBQUERY <i>N</i> .		Page 9-52
Correlated Subquery.	The subquery is correlated.	Page 9-52
Non-correlated Subquery.	The subquery is not correlated.	Page 9-52
Subquery under an IN predicate.	The subquery is introduced by in .	Page 9-53
Subquery under an ANY predicate.	The subquery is introduced by any .	Page 9-53
Subquery under an ALL predicate.	The subquery is introduced by all .	Page 9-53
Subquery under an EXISTS predicate.	The subquery is introduced by exists .	Page 9-53
Subquery under an EXPRESSION predicate.	The subquery is introduced by an expression, or the subquery is in the select list.	Page 9-53
Evaluate Grouped <ONCE, ONCE-UNIQUE or ANY> AGGREGATE	The subquery uses an internal aggregate.	Page 9-55
or		
Evaluate Ungrouped <ONCE, ONCE-UNIQUE or ANY> AGGREGATE		Page 9-54
EXISTS TABLE: nested iteration	The query includes an exists , in , or any clause, and the subquery is flattened into a join.	Page 9-58

For information about how Adaptive Server optimizes certain types of subqueries by materializing results or by flattening the queries to joins, see “Optimizing Subqueries” on page 8-28. For basic information on subqueries, subquery types, and the meaning of the

subquery predicates, see Chapter 5, “Subqueries: Using Queries Within Other Queries” in the *Transact-SQL User’s Guide*.

Output for Flattened or Materialized Subqueries

Certain forms of subqueries can be processed more efficiently when:

- The query is flattened into a join query.
- The subquery result set is materialized as a first step, and the results are used in a second step with the rest of the outer query.

When the optimizer chooses one of these strategies, the query is not processed as a subquery, so you will not see the special subquery message delimiters. The following sections describe `showplan` output for flattened and materialized queries.

Flattened Queries

When subqueries are flattened into existence joins, the output looks like normal `showplan` output for a join, with the possible exception of the message “EXISTS TABLE: nested iteration.”

This message indicates that instead of the normal join processing, which looks for every row in the table that matches the join column, Adaptive Server uses an existence join and returns TRUE as soon as the first qualifying row is located. For more information on subquery flattening, see “Flattening in, any, and exists Subqueries” on page 8-28.

Adaptive Server flattens the following subquery into an existence join:

```
select title
from titles
where title_id in
    (select title_id
     from titleauthor)
and title like "A Tutorial%"
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is SELECT.

FROM TABLE

titles

Nested iteration.

Index : title_ix

Ascending scan.

Positioning by key.

Keys are:

title

Using I/O Size 2 Kbytes.

With LRU Buffer Replacement Strategy.

FROM TABLE

titleauthor

EXISTS TABLE : nested iteration.

Index : titleid_auid_ix

Ascending scan.

Positioning by key.

Index contains all needed columns. Base table will
not be read.

Keys are:

title_id

Using I/O Size 2 Kbytes.

With LRU Buffer Replacement Strategy.

Materialized Queries

When Adaptive Server materializes subqueries, the query is reformulated into two steps:

1. The first step stores the results of the subquery in an internal variable or in a worktable
2. The second step uses the internal variable results or the worktable results in the outer query.

This query materializes the subquery into a worktable:

```
select type, title_id
from titles
where total_sales in (select max(total_sales)
                      from sales_summary
                      group by type)
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is SELECT (into Worktable1).

GROUP BY

Evaluate Grouped MAXIMUM AGGREGATE.

FROM TABLE

sales_summary

Nested iteration.

Table Scan.

Ascending scan.

Positioning at start of table.

Using I/O Size 2 Kbytes.

With LRU Buffer Replacement Strategy.

TO TABLE

Worktable1.

STEP 2

The type of query is SELECT.

FROM TABLE

titles

Nested iteration.

Table Scan.

Ascending scan.

Positioning at start of table.

Using I/O Size 2 Kbytes.

With LRU Buffer Replacement Strategy.

FROM TABLE

Worktable1.

EXISTS TABLE : nested iteration.

Table Scan.

Ascending scan.

Positioning at start of table.

Using I/O Size 2 Kbytes.

With LRU Buffer Replacement Strategy.

The showplan message “EXISTS TABLE: nested iteration,” near the end of the output, shows that Adaptive Server has performed an existence join.

Structure of Subquery *showplan* Output

Whenever a query contains subqueries that are not flattened or materialized:

- The *showplan* output for the outer query appears first. It includes the message “Run subquery *N* (at nesting level *N*)”, indicating the point in the query processing where the subquery executes.
- For each nesting level, the query plans at that nesting level are introduced by the message “NESTING LEVEL *N* SUBQUERIES FOR STATEMENT *N*.”
- The plan for each subquery is introduced by the message “QUERY PLAN FOR SUBQUERY *N* (at line *N*)”, and the end of its plan is marked by the message “END OF QUERY PLAN FOR SUBQUERY *N*.” This section of the output includes information about:
 - The type of query (correlated or uncorrelated)
 - The predicate type (IN, ANY, ALL, EXISTS, or EXPRESSION)

The output structure is shown in Figure 9-1, using the *showplan* output from this query:

```
select title_id
from titles
where total_sales > all (select total_sales
                        from titles
                        where type = 'business')
```

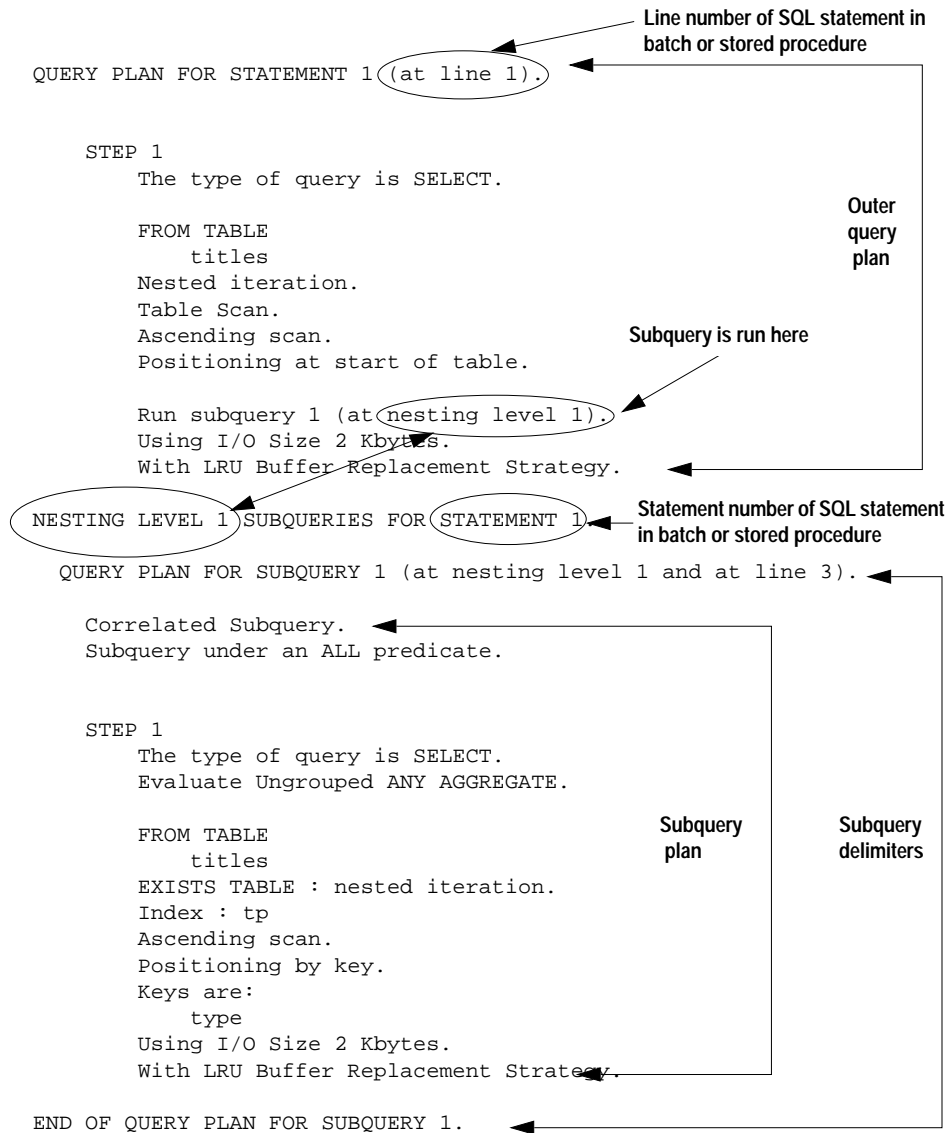


Figure 9-1: Subquery showplan output structure

Subquery Execution Message

Run subquery *N* (at nesting level *N*).

This message shows the place in the execution of the outer query where the subquery execution takes place. Adaptive Server executes the subquery at the point in the outer query where the optimizer finds it should perform best.

The actual plan output for this subquery appears later under the blocks of output for the subquery's nesting level. The first variable in this message is the subquery number; the second variable is the subquery nesting level.

Nesting Level Delimiter Message

NESTING LEVEL *N* SUBQUERIES FOR STATEMENT *N*.

This message introduces the **showplan** output for all the subqueries at a given nesting level. The maximum nesting level is 16.

Subquery Plan Start Delimiter

QUERY PLAN FOR SUBQUERY *N* (at line *N*).

This statement introduces the **showplan** output for a particular subquery at the nesting level indicated by the previous NESTING LEVEL message.

Adaptive Server provides line numbers to help you match query output to your input.

Subquery Plan End Delimiter

END OF QUERY PLAN FOR SUBQUERY *N*.

This statement marks the end of the query plan for a particular subquery.

Type of Subquery

Correlated Subquery.

Non-correlated Subquery.

Every subquery is either correlated or noncorrelated. **showplan** evaluates the type of subquery and, if the subquery is correlated,

returns the message “Correlated Subquery.” Noncorrelated subqueries are usually materialized, so their **showplan** output does not include the normal subquery **showplan** messages.

A correlated subquery references a column in a table that is listed in the **from** list of the outer query. The subquery’s reference to the column is in the **where** clause, the **having** clause, or the select list of the subquery. A noncorrelated subquery can be evaluated independently of the outer query.

Subquery Predicates

Subquery under an **IN** predicate.

Subquery under an **ANY** predicate.

Subquery under an **ALL** predicate.

Subquery under an **EXISTS** predicate.

Subquery under an **EXPRESSION** predicate.

Subqueries introduced by **in**, **any**, **all**, or **exists** are quantified predicate subqueries. Subqueries introduced by **>**, **>=**, **<**, **<=**, **=**, **!=** are expression subqueries.

Internal Subquery Aggregates

Certain types of subqueries require special internal aggregates, as listed in Table 9-6. Adaptive Server generates these aggregates internally—they are not part of Transact-SQL syntax and cannot be included in user queries.

Table 9-6: Internal subquery aggregates

Subquery Type	Aggregate	Effect
Quantified predicate	ANY	Returns 0 or 1 to the outer query.
Expression	ONCE	Returns the result of the subquery. Raises error 512 if the subquery returns more than one value.
Subquery containing distinct	ONCE-UNIQUE	Stores the first subquery result internally and compares each subsequent result to the first. Raises error 512 if a subsequent result differs from the first.

Grouped or Ungrouped Messages

The message “Grouped” appears when the subquery includes a **group by** clause and computes the aggregate for a group of rows.

The message “Ungrouped” appears when the subquery does not include a **group by** clause and computes the aggregate for all rows in the table that satisfy the correlation clause.

Quantified Predicate Subqueries and the ANY Aggregate

Evaluate Grouped ANY AGGREGATE.

Evaluate Ungrouped ANY AGGREGATE.

All quantified predicate subqueries that are not flattened use the internal ANY aggregate. Do not confuse this with the **any** predicate that is part of SQL syntax.

The subquery returns 1 when a row from the subquery satisfies the conditions of the subquery predicate. It returns 0 to indicate that no row from the subquery matches the conditions.

For example:

```
select type, title_id
from titles
where price > all
  (select price
   from titles
   where advance < 15000)
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is SELECT.

FROM TABLE
titles

Nested iteration.

Table Scan.

Ascending scan.

Positioning at start of table.

Run subquery 1 (at nesting level 1).

Using I/O Size 2 Kbytes.

With LRU Buffer Replacement Strategy.

NESTING LEVEL 1 SUBQUERIES FOR STATEMENT 1.

QUERY PLAN FOR SUBQUERY 1 (at nesting level 1 and at line 4).

Correlated Subquery.
Subquery under an ALL predicate.

STEP 1

The type of query is SELECT.
Evaluate Ungrouped ANY AGGREGATE.

FROM TABLE
 titles
 EXISTS TABLE : nested iteration.
 Table Scan.
 Ascending scan.
 Positioning at start of table.
 Using I/O Size 2 Kbytes.
 With LRU Buffer Replacement Strategy.

END OF QUERY PLAN FOR SUBQUERY 1.

Expression Subqueries and the ONCE Aggregate

Evaluate Ungrouped ONCE AGGREGATE.

Evaluate Grouped ONCE AGGREGATE.

Expression subqueries return only a single value. The internal ONCE aggregate checks for the single result required by an expression subquery.

This query returns one row for each title that matches the like condition:

```
select title_id, (select city + " " + state
                  from publishers
                  where pub_id = t.pub_id)
from titles t
where title like "Computer%"
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is SELECT.

FROM TABLE

titles

t

Nested iteration.

Index : title_ix

Ascending scan.

Positioning by key.

Keys are:

title

Run subquery 1 (at nesting level 1).

Using I/O Size 2 Kbytes.

With LRU Buffer Replacement Strategy.

NESTING LEVEL 1 SUBQUERIES FOR STATEMENT 1.

QUERY PLAN FOR SUBQUERY 1 (at nesting level 1 and at line 1).

Correlated Subquery.

Subquery under an EXPRESSION predicate.

STEP 1

The type of query is SELECT.

Evaluate Ungrouped ONCE AGGREGATE.

FROM TABLE

publishers

Nested iteration.

Index : pub_id_ix

Ascending scan.

Positioning by key.

Keys are:

pub_id

Using I/O Size 2 Kbytes.

With LRU Buffer Replacement Strategy.

END OF QUERY PLAN FOR SUBQUERY 1.

Subqueries with *distinct* and the ONCE-UNIQUE Aggregate

Evaluate Grouped ONCE-UNIQUE AGGREGATE.

Evaluate Ungrouped ONCE-UNIQUE AGGREGATE.

When the subquery includes *distinct*, the ONCE-UNIQUE aggregate indicates that duplicates are being eliminated:

```
select pub_name from publishers
where pub_id =
(select distinct titles.pub_id from titles
 where publishers.pub_id = titles.pub_id
 and price > $1000)
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is SELECT.

FROM TABLE

publishers

Nested iteration.

Table Scan.

Ascending scan.

Positioning at start of table.

Run subquery 1 (at nesting level 1).

Using I/O Size 2 Kbytes.

With LRU Buffer Replacement Strategy.

NESTING LEVEL 1 SUBQUERIES FOR STATEMENT 1.

QUERY PLAN FOR SUBQUERY 1 (at nesting level 1 and at line 3).

Correlated Subquery.

Subquery under an EXPRESSION predicate.

STEP 1

The type of query is SELECT.

Evaluate Ungrouped ONCE-UNIQUE AGGREGATE.

FROM TABLE

titles

```
Nested iteration.  
Index : comp_i  
Ascending scan.  
Positioning by key.  
Keys are:  
    price  
Using I/O Size 2 Kbytes.  
With LRU Buffer Replacement Strategy.
```

```
END OF QUERY PLAN FOR SUBQUERY 1.
```

Existence Join Message

```
EXISTS TABLE: nested iteration
```

This message indicates a special form of nested iteration. In a regular nested iteration, the entire table or its index is searched for qualifying values. In an existence test, the query can stop the search as soon as it finds the first matching value.

The types of subqueries that can produce this message are:

- Subqueries that are flattened to existence joins
- Subqueries that perform existence tests

Subqueries That Perform Existence Tests

There are several ways an existence test can be written in Transact-SQL, such as `exists`, `in`, or `=any`. These queries are treated as if they were written with an `exists` clause. The following example demonstrates the `showplan` output with an existence test. This query cannot be flattened because the outer query contains `or`.

```
select au_lname, au_fname  
from authors  
where exists  
    (select *  
     from publishers  
     where authors.city = publishers.city)  
or city = "New York"
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is SELECT.

FROM TABLE

authors

Nested iteration.

Table Scan.

Ascending scan.

Positioning at start of table.

Run subquery 1 (at nesting level 1).

Using I/O Size 2 Kbytes.

With LRU Buffer Replacement Strategy.

NESTING LEVEL 1 SUBQUERIES FOR STATEMENT 1.

QUERY PLAN FOR SUBQUERY 1 (at nesting level 1 and at line 4).

Correlated Subquery.

Subquery under an EXISTS predicate.

STEP 1

The type of query is SELECT.

Evaluate Ungrouped ANY AGGREGATE.

FROM TABLE

publishers

EXISTS TABLE : nested iteration.

Table Scan.

Ascending scan.

Positioning at start of table.

Using I/O Size 2 Kbytes.

With LRU Buffer Replacement Strategy.

END OF QUERY PLAN FOR SUBQUERY 1.

10

Advanced Optimizing Techniques

This chapter describes query processing options that affect the optimizer's choice of join order, index, I/O size and cache strategy.

This chapter contains the following sections:

- Why Special Optimizing Techniques May Be Needed 10-1
- Specifying Optimizer Choices 10-2
- Specifying Table Order in Joins 10-3
- Increasing the Number of Tables Considered by the Optimizer 10-8
- Specifying an Index for a Query 10-8
- Specifying I/O Size in a Query 10-10
- Specifying the Cache Strategy 10-13
- Controlling Large I/O and Cache Strategies 10-14
- Suggesting a Degree of Parallelism for a Query 10-15
- Tuning with dbcc traceon 302 10-17

Why Special Optimizing Techniques May Be Needed

If you have turned to this chapter without fully understanding the materials presented in earlier chapters of this book, be careful when you use the tools described in this chapter. Some of these tools allow you to override the decisions made by Adaptive Server's optimizer and can have an extreme negative effect on performance if they are misused. You need to understand their impact on the performance of your individual query and the possible implications for overall performance.

Adaptive Server's advanced, cost-based optimizer produces excellent query plans in most situations. But there are times when the optimizer does not choose the proper index for optimal performance or chooses a suboptimal join order, and you need to control the access methods for the query. The options described in this chapter allow you that control.

In addition, while you are tuning, you want to see the effects of a different join order, I/O size, or cache strategy. Some of these options

let you specify query processing or access strategy without costly reconfiguration.

Adaptive Server provides tools and query clauses that affect query optimization and advanced query analysis tools that let you understand why the optimizer makes the choices that it does.

► *Note*

This chapter suggests workarounds for certain optimization problems. If you experience these types of problems, call Sybase Technical Support.

Specifying Optimizer Choices

Adaptive Server lets you specify these optimization choices:

- The order of tables in a join
- The number of tables evaluated at one time during join optimization
- The index used for a table access
- The I/O size
- The cache strategy
- The degree of parallelism

In a few cases, the optimizer fails to choose the best plan. In some of these cases, the plan it chooses is only slightly more expensive than the “best” plan, so you need to weigh the cost of maintaining these forced choices against the slower performance of a less than optimal plan.

The commands to specify join order, index, I/O size, or cache strategy, coupled with the query-reporting commands like `statistics io` and `showplan`, can help you determine why the optimizer makes its choices.

◆ **WARNING!**

Use the options described in this chapter with caution. The forced query plans may be inappropriate in some situations and may cause very poor performance. If you include these options in your applications, be sure to check their query plans, I/O statistics, and other performance data regularly.

These options are generally intended for use as tools for tuning and experimentation, not as long-term solutions to optimization problems.

Specifying Table Order in Joins

Adaptive Server optimizes join orders in order to minimize I/O. In most cases, the order that the optimizer chooses does not match the order of the `from` clauses in your `select` command. To force Adaptive Server to access tables in the order they are listed, use the command:

```
set forceplan [on|off]
```

The optimizer still chooses the best access method for each table. If you use `forceplan`, specifying a join order, the optimizer may use different indexes on tables than it would with a different table order, or it may not be able to use existing indexes.

You might use this command as a debugging aid if other query analysis tools lead you to suspect that the optimizer is not choosing the best join order. Always verify that the order you are forcing reduces I/O and logical reads by using `set statistics io on` and comparing I/O with and without `forceplan`.

If you use `forceplan`, your routine performance maintenance checks should include verifying that the queries and procedures that use it still require the option to improve performance.

You can include `forceplan` in the text of stored procedures.

forceplan example

This example is executed with these indexes on the tables in *pubtune*:

- Unique, nonclustered index on *titles(title)*
- Unique, clustered index on *authors(au_id)*
- Unique, nonclustered index on *titleauthor(au_id, title_id)*

Without `forceplan`, this query:

```
select title, au_lname
from titles t, authors a, titleauthor ta
where t.title_id = ta.title_id
and a.au_id = ta.au_id
and title like "Computer%"
```

joins the tables with the join order *titles–titleauthor–authors*, the join order that the optimizer has chosen as the least costly.

Here is the **showplan** output for the unforced query:

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is SELECT.

FROM TABLE

titles

Nested iteration.

Index : title_ix

Ascending scan.

Positioning by key.

Keys are:

title

Using I/O Size 2 Kbytes.

With LRU Buffer Replacement Strategy.

FROM TABLE

titleauthor

Nested iteration.

Index : ta_au_tit_ix

Ascending scan.

Positioning at index start.

Index contains all needed columns. Base table will not
be read.

Using I/O Size 2 Kbytes.

With LRU Buffer Replacement Strategy.

FROM TABLE

authors

Nested iteration.

Using Clustered Index.

Index : au_id_ix

Ascending scan.

Positioning by key.

Keys are:

au_id

Using I/O Size 2 Kbytes.

With LRU Buffer Replacement Strategy.

statistics io for the query shows a total of 163 (90 + 50 + 23) physical
reads and 792 (624 + 93 + 75) logical reads:

```

Table: titles  scan count 1, logical reads: (regular=624 apf=0
total=624), physical reads (regular=8 apf=82 total=90), apf IOs used=82
Table: authors  scan count 34, logical reads: (regular=93 apf=0
total=93), physical reads: (regular=50 apf=0 total=50), apf IOs used=0
Table: titleauthor  scan count 25, logical reads: (regular=75 apf=0
total=75), physical reads: (regular=23 apf=0 total=23), apf IOs used=0
Total actual I/O cost for this command: 4518.
Total writes for this command: 0

```

The total I/O cost for the query is 4518. (This value is printed only when the configuration parameter `allow resource limits` is set to 1.)

If you use `forceplan`, the optimizer chooses a reformatting strategy on *titleauthor*, resulting in this `showplan` report:

QUERY PLAN FOR STATEMENT 1(at line 1).

STEP 1

```

The type of query is INSERT.
The update mode is direct.
Worktable1 created for REFORMATTING.

```

```

FROM TABLE
    titleauthor
Nested iteration.
Index : ta_au_tit_ix
Ascending scan.
Positioning at index start.
Index contains all needed columns. Base table will not
be read.
Using I/O Size 2 Kbytes.
With LRU Buffer Replacement Strategy.
TO TABLE
    Worktable1.

```

STEP 2

```

The type of query is SELECT.

FROM TABLE
    titles
Nested iteration.
Index : title_ix
Ascending scan.
Positioning by key.
Keys are:
    title
Using I/O Size 2 Kbytes.
With LRU Buffer Replacement Strategy.

```

```

FROM TABLE
  authors
Nested iteration.
Table Scan.
Ascending scan.
Positioning at start of table.
Using I/O Size 2 Kbytes.
With LRU Buffer Replacement Strategy.

```

```

FROM TABLE
  Worktable1.
Nested iteration.
Using Clustered Index.
Ascending scan.
Positioning by key.
Using I/O Size 2 Kbytes.
With LRU Buffer Replacement Strategy.

```

statistics io reports the consequences of the extra work created by the poor use of forceplan:

```

Table:titles scan count 1, logical reads:(regular=624 apf=0 total=624),
physical reads: (regular=8 apf=82 total=90), apf IOs used=82
Table: authors scan count 25, logical reads:(regular=5575 apf=0
total=5575), physical reads:(regular=8 apf=26 total=34), apf IOs used=26
Table: titleauthor scan count 1, logical reads: (regular=92 apf=0
total=92), physical reads: (regular=8 apf=14 total=22), apf IOs used=14
Table: Worktable1 scan count 125000, logical reads: (regular=383131
apf=0 total=383131), physical reads:(regular=0 apf=0 total=0), apf IOs
used=0
Total actual I/O cost for this command: 781472.
Total writes for this command: 109

```

Figure 10-1 shows the sequence of the joins and the number of scans required for each query plan.

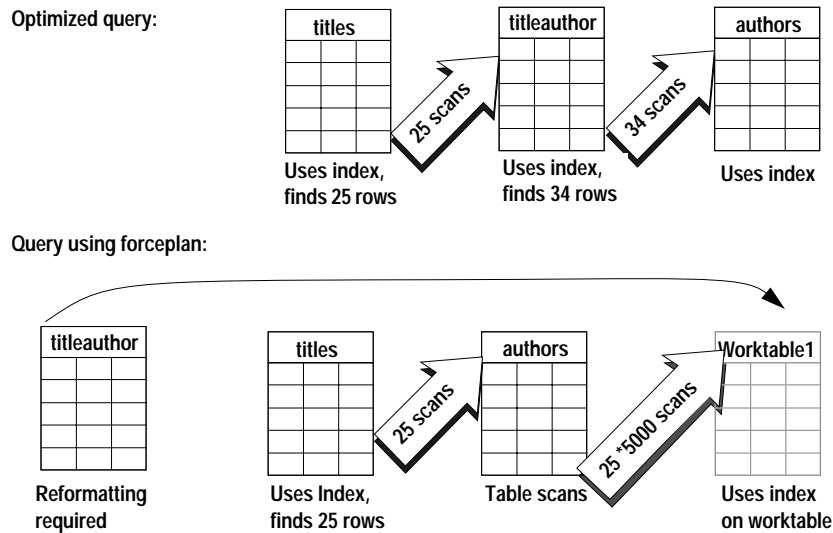


Figure 10-1: Extreme negative effects of using forceplan

Risks of Using *forceplan*

Forcing join order has these risks:

- Misuse can lead to extremely expensive queries.
- It requires maintenance. You must regularly check queries and stored procedures that include *forceplan*. Also, future releases of Adaptive Server may eliminate the problems which led you to incorporate index forcing, so all queries using forced query plans need to be checked each time a new release is installed.

Things to Try Before Using *forceplan*

As the preceding example shows, specifying the join order can be risky. Here are options to try before using *forceplan*:

- Check *showplan* output to determine whether index keys are used as expected.
- Use *dbcc traceon(302)* to look for other optimization problems.
- Be sure that *update statistics* been run on the index recently.

- If the query joins more than four tables, use `set table count` to see if it results in an improved join order. See “Increasing the Number of Tables Considered by the Optimizer” on page 10-8.

Increasing the Number of Tables Considered by the Optimizer

Adaptive Server optimizes joins by considering permutations of four tables at a time, as described in “Optimizing Joins” on page 8-13. If you suspect that an incorrect join order is being chosen for a query that joins more than four tables, you can use the `set table count` option to increase the number of tables that are considered at the same time. The syntax is:

```
set table count int_value
```

The maximum value is 8; the minimum value is 1. As you decrease the value, you reduce the chance that the optimizer will consider all the possible join orders.

Increasing the number of tables considered at one time during join ordering can greatly increase the time it takes to optimize a query.

With Adaptive Server’s default four-at-a-time optimization, it takes 3,024 permutations to consider all the join possibilities. With eight-at-a-time optimization, it takes 40,320 permutations.

Since the time it takes to optimize the query is increased with each additional table, the `set table count` option is most useful when the execution savings from improved join order outweighs the extra optimizing time.

Use `statistics time` to check parse and compile time and `statistics io` to verify that the improved join order is reducing physical and logical I/O.

If increasing the `table count` produces an improvement in join optimization, but increases the CPU time unacceptably, rewrite the `from` clause in the query, specifying the tables in the join order indicated by `showplan` output, and use `forceplan` to run the query. Your routine performance maintenance checks should include verifying that the join order you are forcing still improves performance.

Specifying an Index for a Query

A special clause, (`index index_name`), for the `select`, `update`, and `delete` statements allows you to specify an index for a particular query. You

can also force a query to perform a table scan by specifying the table name. The syntax is:

```
select select_list
  from table_name [table_alias]
      (index {index_name | table_name } )
      [, table_name ...]
  where ...

delete table_name
  from table_name [table_alias]
      (index {index_name | table_name } ) ...

update table_name set col_name = value
  from table_name [table_alias]
      (index {index_name | table_name } ) ...
```

For example:

```
select pub_name, title
  from publishers p, titles t (index date_type)
  where p.pub_id = t.pub_id
        and type = "business"
        and pubdate > "1/1/93"
```

Specifying an index in a query can be helpful when you suspect that the optimizer is choosing a suboptimal query plan. When you use this option:

- Always check statistics for the query to see whether the index you choose requires less I/O than the optimizer's choice.
- Be sure to test a full range of valid values for the query clauses, especially if you are tuning range queries, since the access methods for these queries are sensitive to the size of the range. In some cases, the skew of values in a table or out-of-date statistics cause the optimizer's apparent failure to use the correct index.

Use this option only after testing to be certain that the query performs better with the specified index option. Once you include this index option in applications, you should check regularly to be sure that the resulting plan is still superior to other choices made by the optimizer.

► **Note**

If a nonclustered index has the same name as the table, specifying a table name causes the nonclustered index to be used. You can force a table scan using `select select_list from tablename (0)`.

Risks of Specifying Indexes in Queries

Specifying indexes has these risks:

- Changes in the distribution of data could make the forced index less efficient than other choices.
- Dropping the index means that all queries and procedures that specify the index print an informational message indicating that the index does not exist. The query is optimized using the best available index or other access method.
- Maintenance costs increase, since all queries using this option need to be checked periodically. Also, future releases of Adaptive Server may eliminate the problems which led you to incorporate index forcing, so all queries using forced indexes should be checked each time a new release is installed.

Things to Try Before Specifying Indexes

Before specifying an index in queries:

- Check `showplan` output for the “Keys are” message to be sure that the index keys are being used as expected.
- Use `dbcc traceon(302)` to look for other optimization problems.
- Be sure that `update statistics` has been run on the index recently.

Specifying I/O Size in a Query

If your Adaptive Server is configured for large I/Os in the default data cache or in named data caches, the optimizer can decide to use large I/O for:

- Queries that scan entire tables
- Range queries using clustered indexes, such as queries using `>`, `<`, `> x` and `< y`, `between`, and like `"charstring%"`
- Queries that use covering nonclustered indexes

In these cases, disk I/O can access up to eight pages simultaneously, if the cache used by the table or index is configured for it.

Each named data cache can have several pools, each with a different I/O size. Specifying the I/O size in a query causes the I/O for that query to take place in the pool that is configured for that size. See

Chapter 9, “Configuring Data Caches,” in the *System Administration Guide* for information on configuring named data caches.

To specify an I/O size that is different from the one chosen by the optimizer, add the `prefetch` specification to the `index` clause of a `select`, `delete`, or `update` statement. The syntax is:

```
select select_list
  from table_name
      (index {index_name | table_name} prefetch size)
  [, table_name ...]
  where ...

delete table_name from table_name
  (index {index_name | table_name } prefetch size)
  ...

update table_name set col_name = value
  from table_name
  (index {index_name | table_name} prefetch size) ...
```

Valid values for `size` are 2, 4, 8, and 16. If no pool of the specified size exists in the data cache used by the object, the optimizer chooses the best available size.

If there is a clustered index on `au_lname`, this query performs 16K I/O while it scans the data pages:

```
select *
  from authors (index au_names prefetch 16)
  where au_lname like "Sm%"
```

If a query normally performs large I/O, and you want to check its I/O statistics with 2K I/O, you can specify a size of 2K:

```
select type, avg(price)
  from titles (index type_price prefetch 2)
  group by type
```

► **Note**

If you are experimenting with large I/O sizes and checking `statistics i/o` for physical reads, you may need to clear pages from the cache so that Adaptive Server will perform physical I/O on the second execution of a query. If the table or index, or its database, is bound to a named data cache, you can unbind and rebind the object. If the query uses the default cache, or if other tables or indexes are bound to the object's cache, you can run queries on other tables that perform enough I/O to push the pages out of the memory pools.

Index Type and Large I/O

To perform large I/O on the data pages of a table, specify either the clustered index name, if one exists, or the table name. To perform large I/O on the leaf-level pages of a nonclustered index (for covered queries, for example), specify the nonclustered index name. Table 10-1 shows the effect of the value for the index name clause on prefetching

Table 10-1: Index name and prefetching

Index Name Parameter	Large I/O Performed On
Table name	Data pages
Clustered index name	Data pages
Nonclustered index name	Leaf pages of nonclustered index

When *prefetch* Specification Is Not Followed

Normally, when you specify an I/O size in a query, the optimizer incorporates the I/O size into the query's plan. However, there are times when the specification cannot be followed, either for the query as a whole or for a single, large I/O request.

Large I/O cannot be used for the query:

- If the cache is not configured for I/O of the specified size. The optimizer substitutes the "best" size available.
- If `sp_cachestrategy` has been used to turn off large I/O for the table or index.

Large I/O cannot be used for a single buffer:

- If any of the pages included in that I/O request are in another pool in the cache.
- If the page is on the first extent in an allocation unit. This extent holds the allocation page for the allocation unit, and only seven data pages.
- If no buffers are available in the pool for the requested I/O size.

Whenever a large I/O cannot be performed, Adaptive Server performs 2K I/O on the specific page or pages in the extent that are needed by the query.

To determine whether the prefetch specification is followed, use `showplan` to display the query plan and `statistics io` to see the results on I/O for the query. The system procedure `sp_sysmon` reports on the large I/Os requested and denied for each cache. See “Data Cache Management” on page 24-65.

set prefetch on

By default, a query uses large I/O whenever a large I/O pool is configured and the optimizer determines that large I/O would reduce the query cost. To disable large I/O during a session, use the command:

```
set prefetch off
```

To reenable large I/O, use the command:

```
set prefetch on
```

If large I/O is turned off for an object using the `sp_cachestrategy` system procedure, `set prefetch on` does not override that setting.

If large I/O is turned off for a session using `set prefetch off`, you cannot override the setting by specifying a prefetch size as part of a `select`, `delete`, or `insert` statement.

The `set prefetch` command takes effect in the same batch in which it is run, so you can include it in a stored procedure to affect the execution of the queries in the procedure.

Specifying the Cache Strategy

For queries that scan a table's data pages or the leaf level of a nonclustered index (covered queries), the Adaptive Server optimizer chooses one of two cache replacement strategies: the fetch-and-discard (MRU) strategy or the LRU strategy. See “Overview of Cache Strategies” on page 3-15 for more information about these strategies.

The optimizer may choose the fetch-and-discard (MRU) strategy for:

- Any query that performs table scans
- A range query that uses a clustered index
- A covered query that scans the leaf level of a nonclustered index
- An inner table in a join, if the inner table is larger than the cache
- The outer table of a join, since it needs to be read only once

You can affect the cache strategy for objects:

- By specifying `lru` or `mru` in a `select`, `update`, or `delete` statement
- By using `sp_cachestrategy` to disable or reenable `mru` strategy

If you specify MRU strategy, and a page is already in the data cache, the page is placed at the MRU end of the cache, rather than at the wash marker.

Specifying the cache strategy affects only data pages and the leaf pages of indexes. Root and intermediate pages always use the LRU strategy.

Specifying Cache Strategy in `select`, `delete`, and `update` Statements

You can use `lru` or `mru` (fetch-and-discard) in a `select`, `delete`, or `update` command to specify the I/O size for the query:

```
select select_list
  from table_name
      (index index_name prefetch size [lru|mru])
  [, table_name ...]
  where ...

delete table_name from table_name (index index_name
                                prefetch size [lru|mru]) ...

update table_name set col_name = value
  from table_name (index index_name
                  prefetch size [lru|mru]) ...
```

This query adds the LRU replacement strategy to the 16K I/O specification:

```
select au_lname, au_fname, phone
  from authors (index au_names prefetch 16 lru)
```

For more information about specifying a prefetch size, see “Specifying I/O Size in a Query” on page 10-10.

Controlling Large I/O and Cache Strategies

Status bits in the `sysindexes` table identify whether a table or an index should be considered for large I/O prefetch or for MRU replacement strategy. By default, both are enabled. To disable or reenable these strategies, use the `sp_cachestrategy` system procedure. The syntax is:

```
sp_cachestrategy dbname , [ownername.]tablename
  [, indexname | "text only" | "table only"
  [, { prefetch | mru }, { "on" | "off"}]]
```

This command turns the large I/O prefetch strategy off for the *au_name_index* of the *authors* table:

```
sp_cachestrategy pubtune, authors, au_name_index,
prefetch, "off"
```

This command reenables MRU replacement strategy for the *titles* table:

```
sp_cachestrategy pubtune, titles, "table only",
mru, "on"
```

Only a System Administrator or the object owner can change or view the cache strategy status of an object.

Getting Information on Cache Strategies

To see the cache strategy that is in effect for a given object, execute `sp_cachestrategy`, including only the database and object name:

```
sp_cachestrategy pubtune, titles
```

object name	index name	large IO	MRU
titles	NULL	ON	ON

`showplan` output shows the cache strategy used for each object, including worktables.

Suggesting a Degree of Parallelism for a Query

The `parallel` and `degree_of_parallelism` extensions to the `from` clause of a `select` command allow users to restrict the number of worker processes used in a scan.

For a parallel partition scan to be performed, the `degree_of_parallelism` must be equal to or greater than the number of partitions. For a parallel index scan, specify any value for the `degree_of_parallelism`.

The syntax for the `select` statement is:

```
select ...
  [from {tablename}
    [( index index_name
      [parallel [degree_of_parallelism | 1 ]]
      [prefetch size] [lru|mru] ) ] ,
  {tablename} [( [index_name]
    [parallel [degree_of_parallelism | 1]]
    [prefetch size] [lru|mru] ) ] ...
```

Table 10-2 shows how to combine the `index` and `parallel` keywords to obtain serial or parallel scans.

Table 10-2: Optimizer hints for serial and parallel execution

To Specify This Type of Scan:	Use This Syntax:
Parallel partition scan	(<code>index <i>tablename</i> parallel <i>N</i></code>)
Parallel index scan	(<code>index <i>index_name</i> parallel <i>N</i></code>)
Serial table scan	(<code>index <i>tablename</i> parallel 1</code>)
Serial index scan	(<code>index <i>index_name</i> parallel 1</code>)
Parallel, with the choice of table or index scan left to the optimizer	(<code>parallel <i>N</i></code>)
Serial, with the choice of table or index scan left to the optimizer	(<code>parallel 1</code>)

You cannot use the `parallel` option if you have disabled parallel processing either at the session level with the `set parallel_degree 1` command or at the server level with the configuration parameter `parallel degree`. The `parallel` option cannot override these settings.

If you specify a *degree_of_parallelism* that is greater than the maximum configured degree of parallelism, Adaptive Server ignores the hint.

The optimizer ignores hints that specify a parallel degree if any of the following conditions is true:

- The `from` clause is used in the definition of a cursor.
- `parallel` is used in the `from` clause within any inner query blocks of a subquery, and the optimizer does not move the table to the outermost query block during subquery flattening.
- The table is a view, a system table, or a virtual table.
- The table is the inner table of an outer join.
- The query specifies `exists`, `min`, or `max` on the table.
- The value for the configuration parameter `max scan parallel degree` is set to 1.
- An unpartitioned clustered index is specified or is the only `parallel` option.
- A nonclustered index is covered.

- The query is processed using the OR strategy. (For an explanation of the OR strategy, see “How or Clauses Are Processed” on page 8-22.)
- The select statement is used for an update or insert.

Query Level *parallel* Clause Examples

To specify the degree of parallelism for a single query, include `parallel` after the table name. This example executes in serial:

```
select * from titles (parallel 1)
```

This example specifies the indexes to be used in the query, and sets the degree of parallelism to 5:

```
select * from titles (index cix parallel 5)
```

To force a table scan, use the table name instead of the index name.

Tuning with *dbcc traceon 302*

The `dbcc traceon(302)` flag can often help you understand why the optimizer makes choices that seem incorrect. It can help you debug queries and help you decide whether to use specific options, like specifying an index or a join order for a particular query. It can also help you choose better indexes for your tables.

`showplan` tells you the final decisions that the optimizer makes about your queries. `dbcc traceon(302)` helps you understand why the optimizer made the choices that it did. When you turn on this trace facility, you eavesdrop on the optimizer as it examines query clauses.

The output from this trace facility is more cryptic than `showplan` output, but it provides more detailed information about optimizer choices. The query cost statistics printed by `dbcc traceon(302)` can help to explain, for example, why a table scan is done rather than an indexed access, why *index1* is chosen rather than *index2*, why the reformatting strategy is applied, and so on. The trace output provides detailed information on the costs the optimizer has estimated for permutations of the tables, search clause, and join clause, as well as how it determined those costs.

Invoking the *dbcc* Trace Facility

Execute the following command from an isql batch followed by the query or stored procedure call you want to examine:

```
dbcc traceon(3604, 302)
```

This is what the trace flags mean:

Trace Flag	Explanation
3604	Directs trace output to the client rather than to the error log
302	Prints trace information on index selection

To turn off the output, use:

```
dbcc traceoff(3604, 302)
```

General Tips for Tuning with *dbcc traceon(302)*

To get helpful output from `dbcc traceon(302)`, you need to be sure that your tests cause the optimizer to make the same decisions that it would make while optimizing queries in your application. You must supply the same parameters and values to your stored procedures or `where` clauses. If the application uses cursors, use cursors in your tests.

If you are using stored procedures, make sure that they are actually being optimized during the trial by executing them with `recompile`.

Checking for Join Columns and Search Arguments

In most situations, Adaptive Server uses only one index per table in a query. This means that the optimizer must often choose between indexes when there are multiple `where` clauses supporting both search arguments and join clauses. The optimizer's first step is to match the search arguments and join clauses to available indexes.

The most important item that you can verify using `dbcc traceon(302)` is that the optimizer is evaluating all possible `where` clauses included in each Transact-SQL statement.

If a clause is not included in the output, then the optimizer has determined it is not a valid search argument or join clause. If you believe your query should benefit from the optimizer evaluating this clause, find out why the clause was excluded, and correct it if

possible. The most common reasons for “non-optimizable” clauses include:

- Data type mismatches
- Use of functions, arithmetic, or concatenation on the column
- Numerics compared against constants that are larger than the definition of the column

See “Search Arguments and Using Indexes” on page 8-9 for more information on requirements for search arguments.

Determining How the Optimizer Estimates I/O Costs

Identifying how the optimizer estimates I/O often leads to the root of the problems and to solutions. You will be able to see when the optimizer uses your distribution page statistics and when it uses default values.

Trace Facility Output

Each set of clauses evaluated by the optimizer is printed and delimited within two lines of asterisks. If you issue an unqualified query with no search arguments or join clause, this step is not included in the output, unless the query is covered (a nonclustered index contains all referenced columns).

Output for each qualified clause looks like this:

```
*****
Entering q_score_index() for table 'name' (objectid obj_id,
varno = varno).
The table has X rows and Y pages.
Scoring the clause_type CLAUSE
      column_name operator [column_name]

<other query specific info explained later>
*****
```

`q_score_index()` is the name of a routine that Adaptive Server runs to cost index choices. It finds the best index to use for a given table and set of clauses. The clauses can be either constant search arguments or join clauses.

Identifying the Table

The first line identifies the table name and its associated object ID. The actual output for the line looks like this:

```
Entering q_score_index() for table 'titles' (objectid 208003772),  
varno = 0
```

The optimizer analyzes all search arguments for all tables in each query, followed by all join clauses for each table in the query.

First, you see `q_score_index()` output for all tables in which the optimizer has found a search clause. The routine numbers the tables in the order in which they were specified in the `from` clause and displays the numbers as the `varno`. It starts numbering with 0 for the first table.

Any search clause not included in this section should be evaluated to determine whether its absence impacts performance.

Following the search clause analysis, `q_score_index()` is called for all tables where the optimizer has found a join clause. As above, any join clause not included in this section should be evaluated to determine whether its absence is impacting performance.

If the table is partitioned, `dbcc traceon(302)` also prints the number of partitions, the size of the largest partition, and the skew (the ratio of the largest partition size to the average partition size):

```
The table has 3 partitions.  
The largest partition has 1328 pages. The skew is  
1.070968.
```

Estimating the Table Size

The next line prints the size of the table in both rows and pages:

```
The table has 5000 rows and 624 pages.
```

These sizes are pulled from the OAM pages where they are periodically maintained. There are some known problems where inaccurate row estimates cause bad query plans, so verify that this is not the cause of your problem.

Identifying the *where* Clause

The next two lines indicate the type of clause and a representation of the clause itself with column names and codes for the operators.

These messages indicate:

- That the optimizer is evaluating a search clause. For search clauses, the comparison is shown like this:

```
Scoring the SEARCH CLAUSE:
    au_fname EQ
```

- That the optimizer is evaluating a join clause. For joins, the comparison is shown like this:

```
Scoring the JOIN CLAUSE:
    au_id EQ au_id
```

All search clauses for all tables are evaluated before any join clauses are evaluated. The operator codes are defined in Table 10-3.

Table 10-3: Operators in dbcc traceon(302) output

Code	Comparison in the Query Clause
EQ	Equality comparisons (=)
LT	Less than comparisons (<)
LE	Less than or equal to comparisons (<=)
GT	Greater than comparisons (>)
GE	Greater than or equal to comparisons (>=)
NE	Not equals (!=)
ISNULL	is null comparison
ISNOTNULL	is not null comparison

Output for Range Queries

If your queries include a range query or clauses that are treated like range queries, they are evaluated in a single analysis to produce an estimate of the number of rows for the range. For example:

```
Scoring the SEARCH CLAUSE:
    au_lname LT
    au_lname GT
```

Range queries include:

- Queries using the `between` clause
- Interval clauses with `and` on the same column name, such as:
`datecol1 >= "1/1/94" and datecol1 < "2/1/94"`
- `like` clauses such as:
`like "k%"`

Specified Indexes

If the query has specified the use of a specific index by including the `index` keyword and the index name in parentheses after the table name in the `from` clause, this is noted in the output:

```
User forces index IndexID.
```

Specifying an index prevents the consideration of other alternatives.

If the I/O size and cache strategy are also included in the query, these messages are printed:

```
User forces data prefetch of 8K
```

```
User forces LRU buffer replacement strategy
```

Calculating Base Cost

The next line of output displays the cost of a table scan for comparison, provided that there is at least one other qualification or index that can be considered. It reports index ID 0 and should match the table size estimate displayed earlier. The line looks like this:

```
Base cost: indid: IndexID rows: rows pages: pages prefetch: <S|N>
I/O size: io_size cacheid: cacheID replace: <LRU | MRU>
```

Here is an example:

```
Base cost: indid: 0 rows: 5000 pages: 624 prefetch: N
I/O size: 2 cacheid: 0 replace: LRU
```

Table 10-4 explains the meaning of each value in the output.

Table 10-4: Base cost output

Output	Meaning
indid	The index ID from <i>sysindexes</i> ; 0 for the table itself.
rows	The number of rows in the table.
pages	The number of pages in the table.
prefetch	Whether prefetch would be considered for the table scan.
I/O size	The I/O size to be used.
cacheid	The ID of the data cache to be used.
replace	The cache replacement strategy to be used, either LRU or MRU.

Verify page and row counts for accuracy. Inaccurate counts can cause bad plans. To get a completely accurate count, use the `set statistics io on` command along with a `select * from tablename` query. In a VLDB (very large database) or in 24x7 shops (applications that must run 24 hours a day, 7 days a week), where that is not practical, you may need to rely on the reasonable accuracy of the `sp_spaceused` system procedure. `dbcc allocation-checking` commands print the object size and correct the values on which `sp_spaceused` and other object-size estimates are based.

Costing Indexes

Next, the optimizer evaluates each useful index for a given clause to determine its cost. First, the optimizer looks for a unique index that is totally qualified—that is, the query contains `where` clauses on each key in the index. If such an index is available, the optimizer immediately knows that only a single row satisfies the clause, and it prints the following line:

```
Unique index_type index found--return rows 1 pages pages
```

The *index_type* is either clustered or nonclustered. There are three possibilities for the number of pages:

- The unique index is clustered. The logical I/O cost is the height of the index tree. In a clustered index, the data pages are the leaf level of the index, so the data page access is included.

- The unique, nonclustered index covers the query. The logical I/O is the height of the index tree. The data page access is not needed and is not counted.
- The unique, nonclustered index does not cover the query. An additional logical I/O is necessary to get from the leaf level of the nonclustered index to the data page, so the logical I/O cost is the height of the nonclustered index, plus one page.

If the index is not unique, then the optimizer determines the cost, in terms of logical I/Os, for the clause. Before doing so, it prints this line:

```
Relop bits are: integer
```

This information can be ignored. It merely restates the comparison operator (=, <, >, and so on) listed in the `q_score_index()` line as an integer bitmap. This information is needed by Sybase Engineering to debug optimizer problems; it has no value for customer-level troubleshooting.

To estimate the I/O cost for each clause, the optimizer has a number of tools available to it, depending on the clause type (search clause or join clause) and the availability of index statistics. For more information, see “Index Statistics” on page 7-39.

Index Statistics Used in *dbcc traceon(302)*

For each index, Adaptive Server keeps a statistical histogram of the indexed column’s data distribution. The distribution table is built automatically, when indexes are created, and is stored with the index. The table is a sampling of the index key values every *N* rows.

N is dependent on the full size of the key (including overhead) and the number of rows in the table. Each sampling is known as a “step.” Since the optimizer knows how many rows exist between steps and the density of keys in the index, it can estimate the number of rows satisfying a clause with reasonable accuracy. See “Index Statistics” on page 7-39 for more information.

Evaluating Statistics for Search Clauses

For search clauses, the optimizer can look up specific values on the distribution page, if these values are known at compile time. In this case, it prints the distribution page number and the number of steps with the following trace output:

```
Qualifying stat page; pgno: page_number steps: steps
```


For atomic datatypes, such as *tinyint*, *smallint*, *int*, *char*, *varchar*, *binary*, and *varbinary*, which are not internally implemented as structures, it prints the constant value the search argument supplied to the optimizer. It looks like this:

Search value: *constant_value*

If the value is implemented as a structure, the following message is printed to indicate that the optimizer does not waste time building the structure's printable representation:

```
*** CAN'T INTERPRET ***
```

Distribution Page Value Matches

If an exact match is found on the distribution page, the following message is printed:

Match found on statistics page

This is followed by information pertaining to the number and location of the step values found on the distribution page. Since the optimizer knows approximately how many rows exist between step values, it uses this information to estimate how many logical I/Os will be performed for this clause. To indicate this information, one of the following messages is displayed:

equal to several rows including 1st or last -use endseveralSC

This indicates that several steps matched the constant, and they were found either at the beginning or at the end of the distribution page.

equal to a single row (1st or last) -use endsingleSC

This indicates that only one step matched the constant, and it was found either at the beginning or at the end of the distribution page.

equal to several rows in middle of page -use midseveralSC

This indicates that several steps matched the constant, and they were found in the middle of the distribution page.

equal to single row in middle of page -use midsingleSC

This indicates that several steps matched the constant, and they were found in the middle of the distribution page.

Values Between Steps or Out of Range

If an exact match of the search value is not found on the distribution page, the optimizer uses different formulas in its statistical estimate.

The computation is based on the relational operator used in the clause, the step number, the number of steps, the number of rows in the table, and the index density. First, the optimizer needs to find the first step value that is less than the search value. In these cases, you see the following message:

```
No steps for search value -qualpage for LT search value finds
```

Depending on whether the search value is outside the first or last step value or contained within the steps, the optimizer prints one of the following messages:

```
value < first step -use outsideSC
```

```
value > last step -use outsideSC
```

The first message indicates that the query's search value is smaller than the first entry on the distribution page. The second message indicates that the query's search value is larger than the last entry on the distribution page. If the constant is a valid value in the table, these messages indicate that you may need to run `update statistics`.

```
value between step K, K+1, K=step_number -use betweenSC
```

This message indicates that the query's search value falls between two steps on the distribution page. You can only confirm here that the step number seems reasonable.

For example, if the step value of "K" is 3, and you suspect that the query's search value should fall near the end of the table, something could be wrong. It would be reasonable then to expect the value of "K" to be larger (that is, nearer the end of the distribution page). This may be another indication that you need to run `update statistics`.

Range Query Messages

For a range query, the trace facility looks up the steps for both the upper and lower bounds of the query. This message appears:

```
Scoring SARG interval, lower bound.
```

After displaying the costing estimates for the lower bound, the net selectivity is calculated and displayed as:

```
Net selectivity of interval: float_value
```

Search Clauses with Unknown Values

A common problem the optimizer faces is that values for search criteria are not known until run time. Common scenarios that make the optimizer unable to use distribution statistics include:

- **where** clauses based on expressions. For example:

```
select *
  from tableName
  where dateColumn >= getdate()
```

- **where** clauses based on local variables. For example:

```
declare @fKey int
select @fKey=lookUpID
  from mainTable
  where pKey = "999"
select *
  from lookUpTable
  where pKey >= @fKey
```

In cases like these, the optimizer tries to make intelligent guesses, based on average values. For example, if a distribution page exists for the index, and the query is an equality comparison, the optimizer uses the density of the index (that is, the average number of duplicates) to estimate the number of rows.

Otherwise, the optimizer uses a “magic” number: it assumes that 10 percent of the table will match an equality comparison, 25 percent of the table will match a closed interval comparison, and 33 percent of the table will match for inequality and open interval comparisons. In these cases, the trace facility prints:

```
SARG is a subbed VAR or expr result or local variable (constat =
number) -use magicSC or densitySC
```

Stored procedures and triggers need special attention to ensure efficient query plans. Many procedures are coded with **where** clauses, based on input parameters. This behavior can cause some difficulty in troubleshooting. Consider a query whose single **where** clause may be very selective under one parameter and return nearly the entire table under another. These types of stored procedures can be difficult to debug, since the procedure cache can potentially have multiple copies in cache, each with a different plan. Since these plans are already compiled, a user may be assigned a plan that may not be appropriate for a particular input parameter.

Another reason for the optimizer being unable to use an index’s distribution table is that the distribution page can be nonexistent. This occurs:

- When an index is created before any data is loaded
- When **truncate table** is used, and then the data is loaded

In these cases, the optimizer uses the above mentioned “magic” numbers to estimate I/O cost, and you see:

```
No statistics page -use magicSC
```

At this point, the selectivity and cost estimates are displayed.

Cost Estimates and Selectivity

For each qualified clause, the dbcc traceon(302) displays:

- The index ID
- The selectivity as a floating-point value
- The cost estimate in both rows and pages

These values are printed as variables in this message:

```
Estimate: indid indexID, selectivity float_val, rows rows pages pages
```

If this clause has no qualifications, but the optimizer finds a nonclustered index that covers the entire query, it identifies the table, since it is not listed in `q_score_index()` section. In this case, you see:

```
Finishing q_score_index() for table table_name (objectid) ID.
```

At this point, the cheapest index is examined, and its costs are displayed:

```
Cheapest index is index IndexID, costing pages pages and  
generating rows rows per scan using no data prefetch (size 2)  
on dcacheid N with [MRU|LRU] replacement
```

This can be somewhat misleading. If there are any nonclustered indexes that match the search arguments in the query, the costs for the cheapest index are printed here, even though a table scan may be used to execute the query. The actual decision on whether to perform a table scan or nonclustered index access is delayed until the join order is evaluated (the next step in the optimization process).

This is because the most accurate costing of a nonclustered index depends on the ratio of physical vs. logical I/O and the amount of cache memory available when that table is chosen. Therefore, if the base cost (table scan cost) printed earlier is significantly less than the “cheapest index” shown here, it is more likely that a table scan will be used. Use `showplan` to verify this.

Estimating Selectivity for Search Clauses

The selectivity for search clauses is printed as the fraction of the rows in the table that are expected to qualify. Therefore, the lower the number, the more selective the search clause and the fewer the rows that are expected to qualify. Search clauses are output as:

```
Search argument selectivity is float_val.
```

Estimating Selectivity for Join Clauses

For joins, the optimizer never looks up specific values on the distribution page. At compile time, the optimizer has no known values for which to search. It needs to make a sophisticated estimate about costing these clauses.

If an index with a distribution page is available, the optimizer uses the density table, which stores the average number of duplicate keys in the index. All leading permutations of the composite key have their density stored, providing accurate information for multicolumn joins.

If no distribution page is available for this index, the optimizer estimates the join selectivity to be 1 divided by the number of rows in the smaller table. This gives a rough estimate of the cardinality of a primary key-foreign key relationship with even data distribution. In both of these cases, the trace facility prints the calculated selectivity and cost estimates, as described below.

The selectivity of the clause is printed last. Join clauses are output as:

```
Join selectivity is float_val.
```

The selectivity for join clauses is output as a float value. The selectivity of the join key is the fraction 1 divided by the value. Therefore, the higher the number selectivity, the more selective the join clause, and the fewer the rows that are expected to qualify.

If a unique index exists on a join clause, the following message is printed:

```
Unique nonclustered index found--return rows 1  
pages 4
```

At this point, the optimizer has evaluated all indexes for this clause and will proceed to optimize the next clause.

11

Transact-SQL Performance Tips

This chapter presents certain types of SQL queries where simple changes in the query can improve performance. This chapter emphasizes queries only and does not focus on schema design.

This chapter contains the following sections:

- “Greater Than” Queries 11-1
- not exists Tests 11-2
- Variables vs. Parameters in where Clauses 11-3
- Count vs. Exists 11-4
- Aggregates 11-5
- or Clauses vs. Unions in Joins 11-5
- Joins and Datatypes 11-6
- Parameters and Datatypes 11-8

These tips are intended as suggestions and guidelines, not absolute rules. You should use the query analysis tools to test the alternate formulations suggested here.

Performance of these queries may change with future releases of Adaptive Server.

“Greater Than” Queries

This query, with an index on *int_col*:

```
select * from table where int_col > 3
```

uses the index to find the first value where *int_col* equals 3, and then scans forward to find the first value that is greater than 3. If there are many rows where *int_col* equals 3, the server has to scan many pages to find the first row where *int_col* is greater than 3.

It is probably much more efficient to write the query like this:

```
select * from table where int_col >= 4
```

This optimization is easier with integers, but more difficult with character strings and floating-point data. You need to know your data.

not exists Tests

In subqueries and if statements, `exists` and `in` perform faster than `not exists` and `not in`, when the values in the `where` clause are not indexed. For `exists` and `in`, Adaptive Server can return TRUE as soon as a single row matches. For the negated expressions, it must examine all values to determine that there are not matches.

In if statements, you can easily avoid `not exists` by rearranging your statement groups between the if portion and the else portion of the code. This `not exists` test may perform slowly:

```
if not exists (select * from table where...)
begin
    /* Statement Group 1 */
end
else
begin
    /* Statement Group 2 */
end
```

You can improve the performance of the query by using `exists` and switching the statement groups:

```
if exists (select * from table where...)
begin
    /* Statement Group 2 */
end
else
begin
    /* Statement Group 1 */
end
```

You can avoid the `not else` in if statements, even without an else clause. Here is an example:

```
if not exists (select * from table where...)
begin
    /* Statement Group */
end
```

This query can be rewritten using `goto` to skip over the statement group:

```
if exists (select * from table where)
begin
    goto exists_label
end
/* Statement group */
exists_label:
```


Variables vs. Parameters in *where* Clauses

The optimizer knows the value of a parameter to a stored procedure at compile time, but it cannot predict the value of a variable that is set in the procedure while it executes. Providing the optimizer with the values of search arguments in the *where* clause of a query can help the optimizer to make better choices—when the optimizer knows the value of a search clause, it can use the distribution steps to predict the number of rows that the query will return. Otherwise, it uses only density statistics or hard-coded values.

Often, the solution to this type of performance problem is to split up a stored procedure so that you set the values of variables in the first procedure and then call the second procedure, passing the variables as parameters. The second procedure can then be optimized correctly.

For example, the optimizer cannot optimize the final *select* in the following procedure, because it cannot know the value of *@x* until execution time:

```
create procedure p
as
  declare @x int
  select @x = col
         from tab where ...
  select *
         from tab2
         where indexed_col = @x
```

When Adaptive Server encounters unknown values, it uses approximations to develop a query plan, based on the operators in the search argument, as shown in Table 11-1.

Table 11-1: Density approximations for unknown search arguments

Operator	Density Approximation
=	Average proportion of duplicates in the column
< or >	33%
between	25%

The following example shows the procedure split into two procedures. The first procedure calls the second one:

```
create procedure base_proc
as
    declare @x int
    select @x = col
           from tab where ...
    exec select_proc @x

create procedure select_proc @x int
as
    select *
           from tab2
           where col2 = @x
```

When the second procedure executes, Adaptive Server knows the value of `@x` and can optimize the `select` statement. Of course, if you modify the value of `@x` in the second procedure before it is used in the `select` statement, the optimizer may choose the wrong plan because it optimizes the query based on the value of `@x` at the start of the procedure. If `@x` has different values each time the second procedure is executed, leading to very different query plans, you may want to use `with recompile`.

Count vs. Exists

Do not use the `count` aggregate in a subquery to do an existence check:

```
select *
    from tab
    where 0 < (select count(*) from tab2 where ...)
```

Instead, use `exists` (or `in`):

```
select *
    from tab
    where exists (select * from tab2 where ...)
```

Using `count` to do an existence check is slower than using `exists`.

When you use `count`, Adaptive Server does not know that you are doing an existence check. It counts all matching values, either by doing a table scan or by scanning the smallest nonclustered index.

When you use `exists`, Adaptive Server knows you are doing an existence check. When it finds the first matching value, it returns `TRUE` and stops looking. The same applies to using `count` instead of `in` or `any`.

or Clauses vs. Unions in Joins

Adaptive Server cannot optimize join clauses that are linked with `or` and it may perform Cartesian products to process the query.

► **Note**

Adaptive Server does not optimize search arguments that are linked with `or`. This description applies only to join clauses.

Adaptive Server can optimize selects with joins that are linked with `union`. The result of `or` is somewhat like the result of `union`, except for the treatment of duplicate rows and empty tables:

- `union` removes all duplicate rows (in a sort step); `union all` does not remove any duplicates. The comparable query using `or` might return some duplicates.
- A join with an empty table returns no rows.

For example, when Adaptive Server processes this query, it must look at every row in one of the tables for each row in the other table:

```
select *
  from tab1, tab2
 where tab1.a = tab2.b
        or tab1.x = tab2.y
```

If you use `union`, each side of the union is optimized separately:

```
select *
  from tab1, tab2
 where tab1.a = tab2.b
union all
select *
  from tab1, tab2
 where tab1.x = tab2.y
```

You can use `union` instead of `union all` if you want to eliminate duplicates, but this eliminates all duplicates. It may not be possible to get exactly the same set of duplicates from the rewritten query.

Aggregates

Adaptive Server uses special optimizations for the `max` and `min` aggregate functions when there is an index on the aggregated column:

- For `min`, it reads the first value on the root page of the index.

- For `max`, it goes directly to the end of the index to find the last row.
- `min` and `max` optimizations are not applied if:
- The expression inside the `max` or `min` function is anything but a column. Compare `max(numeric_col*2)` and `max(numeric_col)*2`, where `numeric_col` has a nonclustered index. The syntax in the first example contains an operation on a column, so the query performs a leaf-level scan of the nonclustered index. The second syntax example uses `max` optimization, because the multiplication is performed on the result of the function.
 - The column inside the `max` or `min` aggregate is not the first column of an index. For nonclustered indexes, it can perform a scan on the leaf level of the index; for clustered indexes, it must perform a table scan.
 - There is another aggregate in the query.
 - There is a `group by` clause.

In addition, the `max` optimization is not applied if there is a `where` clause.

If you have an optimizable `max` or `min` aggregate, you should get much better performance by putting it in a query that is separate from other aggregates.

For example:

```
select max(price), min(price)
      from titles
```

results in a full scan of `titles`, even if there is an index on `colx`.

Try rewriting the query as:

```
select max(price)
      from titles
select min(price)
      from titles
```

Adaptive Server uses the index once for each of the two queries, rather than scanning the entire table.

Joins and Datatypes

When joining between two columns of different datatypes, one of the columns must be converted to the type of the other. The *Adaptive Server Reference Manual* shows the hierarchy of types. The column whose type is lower in the hierarchy is the one that is converted.

If you are joining tables with incompatible types, one of them can use an index, but the query optimizer cannot choose an index on the column that it converts. For example:

```
select *
  from small_table, large_table
 where smalltable.float_column =
        large_table.int_column
```

In this case, Adaptive Server converts the integer column to *float*, because *int* is lower in the hierarchy than *float*. It cannot use an index on *large_table.int_column*, although it can use an index on *smalltable.float_column*.

Null vs. Not Null Character and Binary Columns

Note that *char null* is really stored as *varchar*, and *binary null* is really *varbinary*. Joining *char not null* with *char null* involves a conversion; the same is true of the binary types. This affects all character and binary types, but does not affect numeric datatypes and datetimes.

It is best to avoid datatype problems in joins by designing the schema accordingly. Frequently joined columns should have the same datatypes, including the acceptance of null values for character and binary types. User-defined datatypes help enforce datatype compatibility.

Forcing the Conversion to the Other Side of the Join

If a join between different datatypes is unavoidable, and it hurts performance, you can force the conversion to the other side of the join.

In the following query, *varchar_column* must be converted to *char*, so no index on *varchar_column* can be used, and *huge_table* must be scanned:

```
select *
  from small_table, huge_table
 where small_table.char_col =
        huge_table.varchar_col
```

Performance would be improved if the index on *huge_table.varchar_col* could be used. Using the *convert* function on the *varchar* column of the small table allows the index on the large table to be used while a table scan is performed on the small table:

```

select *
  from small_table, huge_table
 where convert(varchar(50),small_table.char_col) =
        huge_table.varchar_col

```

Be careful with numeric data. This tactic can change the meaning of the query. This query compares integers and floating-point numbers:

```

select *
  from tab1, tab2
 where tab1.int_column = tab2.float_column

```

In this example, *int_column* is converted to *float*, and any index on *int_column* cannot be used. If you insert this conversion to force the index access to *tab1*:

```

select *
  from tab1, tab2
 where tab1.int_col = convert(int, tab2.float_col)

```

the query will not return the same results as the join without the `convert`. For example, if *int_column* is 4, and *float_column* is 4.2, the original query implicitly converts 4 to 4.0000, which does not match 4.2. The query with the `convert` converts 4.2 to 4, which does match.

The query can be salvaged by adding this self-join:

```

and tab2.float_col = convert(int, tab2.float_col)

```

This assumes that all values in *tab2.float_col* can be converted to *int*.

Parameters and Datatypes

The query optimizer can use the values of parameters to stored procedures to help determine costs.

If a parameter is not of the same type as the column in the `where` clause to which it is being compared, Adaptive Server has to convert the parameter.

The optimizer cannot use the value of a converted parameter.

Make sure that parameters are of the same type as the columns they are compared to.

For example:

```

create proc p @x varchar(30)
as
  select *
    from tab
   where char_column = @x

```

may get a less optimal query plan than:

```
create proc p @x char(30)
as
  select *
    from tab
   where char_column = @x
```

Remember that *char null* is really *varchar*, and *binary null* is really *varbinary*.

12

Cursors and Performance

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 are locking at the page and at the table level, network resources, and overhead of processing instructions.

This chapter contains the following sections:

- What Is a Cursor? 12-1
- Resources Required at Each Stage 12-4
- Cursor Modes: Read-Only and Update 12-6
- Index Use and Requirements for Cursors 12-6
- Comparing Performance With and Without Cursors 12-7
- Locking with Read-Only Cursors 12-10
- Locking with Update Cursors 12-11
- Isolation Levels and Cursors 12-12
- Partitioned Heap Tables and Cursors 12-13
- Optimizing Tips for Cursors 12-13

What Is a Cursor?

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. Table 12-1 shows a cursor accessing the *authors* table.

Cursor with <code>select * from authors</code> where state = 'KY'		Result set		
	➔ A978606525	Marcello	Duncan	KY
	➔ A937406538	Carton	Nita	KY
Programming can:				
- Examine a row	➔ A1525070956	Porczyk	Howard	KY
- Take an action based on row values	➔ A913907285	Bier	Lane	KY

Figure 12-1: Cursor example

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 vs. Row-Oriented Programming

SQL was not conceived as a row-oriented language—it 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 when needed, so when you need the flexibility, use them.

Figure 12-2 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.

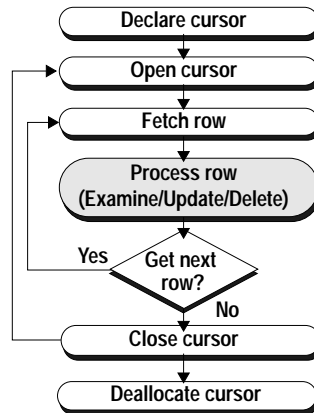


Figure 12-2: Cursor flowchart

Cursors: A Simple Example

Here is a simple example of a cursor with the “Process Rows” step from Table 12-2 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 declare 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 perhaps page locks. When you fetch a row, there is a page lock on the page that stores the row, locking out updates by other processes. If you fetch multiple rows, there is a page lock on each page that contains a fetched row. Table 12-3 shows the duration of locks during cursor operations.

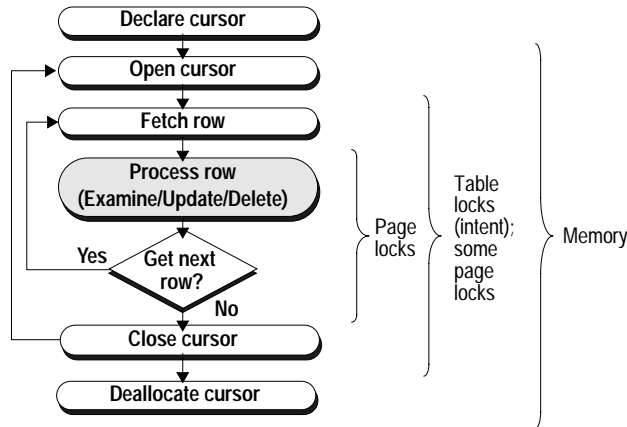


Figure 12-3: Resource use by cursor statement

The memory resource descriptions in Figure 12-3 and Table 12-1 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 12-5.

Table 12-1: Locks and memory use for isql and Client-Library client cursors

Cursor Command	Resource Use
<code>declare cursor</code>	When you declare a cursor, Adaptive Server allocates memory to the cursor and to store the query plan that is generated. The size of the query plan depends on the <code>select</code> statement, but it generally ranges from one to two pages.
<code>open</code>	When you open a cursor, Adaptive Server starts processing the <code>select</code> statement. 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) and, if there are subqueries or joins, it acquires page locks on the outer table(s).
<code>fetch</code>	When you execute a <code>fetch</code> , Adaptive Server acquires a page lock, gets the row(s) required and reads specified values into the cursor variables or sends the row to the client. The page lock is held until a <code>fetch</code> moves the cursor off the page or until the cursor is closed. The page lock is either a shared or an update page lock, depending on how the cursor is written.
<code>close</code>	When you close a cursor, Adaptive Server releases the shared locks and some of the memory allocation. You can open the cursor again, if necessary.
<code>deallocate cursor</code>	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 12-1 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 pre-compiler 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: Read-Only and Update

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 locks. It 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 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.

Read-Only vs. Update

Specify the cursor mode when you declare the cursor. Note that 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

Any index can be used for read-only cursors. They should produce the same query plan as the select statement outside of a cursor. The index requirements for updatable cursors are rather specific, and updatable cursors may produce 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. But a unique index is not required.
- If the cursor is declared for update **without** a `for update of list`, a unique index is required. 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. 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.

A query that you use without a cursor may use very different indexes if you include the same query in a cursor.

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

  /* 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 vs. Non-Cursor Performance Comparison

Table 12-2 shows actual statistics gathered against a 5000-row table. Note that the cursor code takes two and one-half times longer, even though it scans the table only once.

Table 12-2: Sample execution times against a 5000-row table

Procedure	Access Method	Time
increase_price	Uses three table scans	2 minutes
increase_price_cursor	Uses cursor, single table scan	5 minutes

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.

Cursor vs. Non-Cursor Performance Explanation

In addition to locking, cursors involve much 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.

When cursors are absolutely necessary, of course they should be used. But they can adversely affect performance.

Cursor performance issues are:

- Locking at the page and table level
- Network resources
- Overhead of processing instructions

Use cursors only if necessary. 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. Execute the code in Figure 12-4, pausing to execute `sp_lock` where the arrows are.

Using `sp_lock`, examine the locks that are in place at each arrow:

```

declare curs1 cursor for
select au_id, au_lname, 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

```

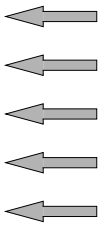


Figure 12-4: Read-only cursors and locking experiment input

Table 12-3 shows the results.

Table 12-3: Locks held on data and index pages by cursors

Event	Data Page
After declare	No cursor-related locks.
After open	Shared intent lock on <i>authors</i> .
After first fetch	Shared intent lock on <i>authors</i> and shared page lock on a page in <i>authors</i> .
After 100 fetches	Shared intent lock on <i>authors</i> and shared page lock on a different page in <i>authors</i> .
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.

Locking with Update Cursors

The next example requires two connections to Adaptive Server.

Open two connections to Adaptive Server, and execute the commands shown in Figure 12-5.

Connection 1	Connection 2
<pre> declare curs2 cursor for select au_id, au_lname from authors where au_id like 'A1%' for update go open curs2 go fetch curs2 go delete from authors where current of curs2 go /* what happens? */ close curs2 go </pre>	<pre> begin tran go select * from authors holdlock where au_id = au_id fetched at left go sp_lock go delete from authors where au_id = same au_id /* what happens? */ </pre>

Figure 12-5: Update cursors and locking experiment input

Update Cursors: Experiment Results

Connection 1, which opens a cursor and fetches a row, gets an update lock on that page, which allows shared locks but not exclusive locks or update locks.

When Connection 2 does a select with holdlock, that works because it just needs a shared lock.

When Connection 1 (the cursor) tries to delete, it needs an exclusive lock but cannot get it, which means it has to wait. When Connection 2 tries to delete, which requires an exclusive lock, it cannot get it either, and deadlock occurs.

Isolation Levels and Cursors

The query plan for a cursor is compiled and optimized at the time it is declared. You cannot declare 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 declare a cursor at isolation level 0 and open or fetch from it at level 1 or 3. Similarly, you cannot declare a cursor at level 1 or 3 and then fetch from it at level 0. Attempts to open or fetch from a cursor at an incompatible level result in 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 at 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
```

If you declare a cursor at level 1 and use `set transaction isolation level` to change the level to 3, you can still open and fetch from the cursor, but it operates at level 1, that is, it does not hold locks on all of the pages that are fetched. The reverse is also true: if you declare the cursor at level 3 and fetch from it at level 1, it continues to hold locks on all of the fetched pages.

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 "How *or* Clauses Are Processed" on page 8-22 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 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 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 12-4 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 the table, the additional locks, or more restrictive locks for the two versions of the **for update** clause are emphasized.

Table 12-4: Effects of **for update** clause and **shared** on cursor locking

Clause	<i>titles</i>	<i>authors</i>	<i>titleauthor</i>
None	sh_page on data	sh_page on data	sh_page on data
for update	updpage on index updpage on data	updpage on index updpage on data	updpage on data
for update of price	updpage on data	sh_page on index sh_page on data	sh_page on data
for update of price + shared	sh_page on data	sh_page index sp_page 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 20-3 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 in order to maintain concurrency and throughput.

If you must be ANSI-compliant, you need to 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 follow the usual advice to keep 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. If you choose 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 connection 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.

Parallel Query Concepts and Tuning

13 Introduction to Parallel Query Processing

This chapter introduces basic concepts and terminology needed for parallel query optimization, parallel sorting, and other parallel query topics, and provides an overview of the commands for working with parallel queries.

This chapter covers the following topics:

- Types of Queries That Can Benefit from Parallel Processing 13-2
- Adaptive Server's Worker Process Model 13-3
- Types of Parallel Data Access 13-6
- Controlling the Degree of Parallelism 13-10
- Commands for Working with Partitioned Tables 13-15
- Balancing Resources and Performance 13-16
- Guidelines for Parallel Query Configuration 13-18
- System Level Impacts 13-23
- When Parallel Query Results Can Differ 13-24

Other chapters that cover specific parallel processing topics in more depth are as follows:

- For details on how the Adaptive Server optimizer determines eligibility and costing for parallel execution, see Chapter 14, "Parallel Query Optimization."
- To understand parallel sorting topics, see Chapter 15, "Parallel Sorting."
- For information on object placement for parallel performance, see "Partitioning Tables for Performance" on page 17-13.
- For information about locking behavior during parallel query processing see Chapter 5, "Locking in Adaptive Server."
- For information on `showplan` messages, see "showplan Messages for Parallel Queries" on page 9-42
- To understand how Adaptive Server uses multiple engines, see Chapter 21, "How Adaptive Server Uses Engines and CPUs."

Types of Queries That Can Benefit from Parallel Processing

When Adaptive Server is configured for parallel query processing, the optimizer evaluates each query to determine whether it is eligible for parallel execution. If it is eligible, and if the optimizer determines that a parallel query plan can deliver results faster than a serial plan, the query is divided into components that process simultaneously. The results are combined and delivered to the client in a shorter period of time than it would take to process the query serially as a single component.

Parallel query processing can improve the performance of the following types of queries:

- select statements that scan large numbers of pages but return relatively few rows, such as:
 - Table scans or clustered index scans with grouped or ungrouped aggregates
 - Table scans or clustered index scans that scan a large number of pages, but have *where* clauses that return only a small percentage of the rows
- select statements that include *union*, *order by* or *distinct*, since these queries can populate worktables in parallel, and can make use of parallel sorting
- select statements where the reformatting strategy is chosen by the optimizer, since these can populate worktables in parallel, and can make use of parallel sorting
- create index statements, and the *alter table...add constraint* clauses that create indexes, unique and primary key
- The *dbcc checkstorage* command

Join queries can use parallel processing on one or more tables.

Commands that return large, unsorted result sets are unlikely to benefit from parallel processing due to network constraints—in most cases, results can be returned from the database faster than they can be merged and returned to the client over the network.

Commands that modify data (*insert*, *update*, and *delete* commands), and cursors do not run in parallel. The inner, nested blocks of queries containing subqueries are never executed in parallel, but the outer block can be executed in parallel.

Decision support system (DSS) queries that access huge tables and return summary information benefit the most from parallel query

processing. The overhead of allocating and managing parallel queries makes parallel execution less effective for online transaction processing (OLTP) queries, which generally access fewer rows and join fewer tables. When a server is configured for parallel processing, only queries that access 20 data pages or more are considered for parallel processing, so most OLTP queries run in serial; those that access more pages are eligible to run in parallel.

Adaptive Server's Worker Process Model

Adaptive Server uses a **coordinating process** and multiple **worker processes** to execute queries in parallel. In some respects, a query that runs in parallel with eight worker processes is much like eight serial queries accessing one-eighth of the table, with the coordinating process supervising the interaction and managing the process of returning results to the client. Each worker process uses approximately the same amount of memory as a user connection. Each worker process runs as a task that must be scheduled on an engine, scans data pages, queues disk I/Os, and performs in many ways like any other task on the server. One major difference is that in last phase of query processing, the coordinating process manages merging the results and returning them to the client, coordinating with worker processes.

Figure 13-1 shows the events that take place during parallel query processing:

1. The client submits a query.
2. The client task assigned to execute the query becomes the coordinating process for parallel query execution.
3. The coordinating process requests four worker processes from the pool of worker processes allocated at Adaptive Server start-up. The coordinating process together with the worker processes is called a **family**.
4. The worker processes execute the query in parallel.
5. The coordinating process returns results produced by all the worker processes to the client.

The serial client shown in the lower-right corner of Figure 13-1 submits a query that is processed serially.

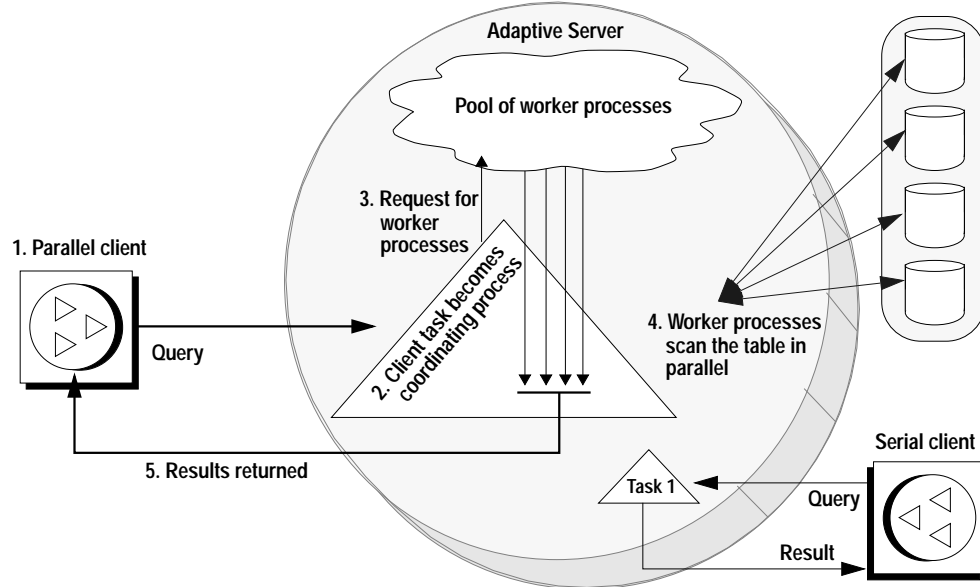


Figure 13-1: Worker process model

During query processing, the tasks are tracked in the system tables by a family ID (*fid*). Each worker process for a family has the same family ID and its own unique server process ID (*spid*). System procedures such as `sp_who` and `sp_lock` display both the *fid* and the *spid* for parallel queries, allowing you to observe the behavior of all processes in a family.

Parallel Query Execution

Figure 13-2 shows how parallel query processing reduces response time over the same query running in serial. A good example of a query that would show this type of activity is a query that contains a `group by` clause. In parallel execution, three worker processes scan the data pages. The times required by each worker process may vary, depending on the amount of data that each process needs to access. Also, a scan can be temporarily blocked due to locks on data pages held by other users. When all of the data has been read, the results

from each worker process are merged into a single result set by the coordinating process and returned to the client.

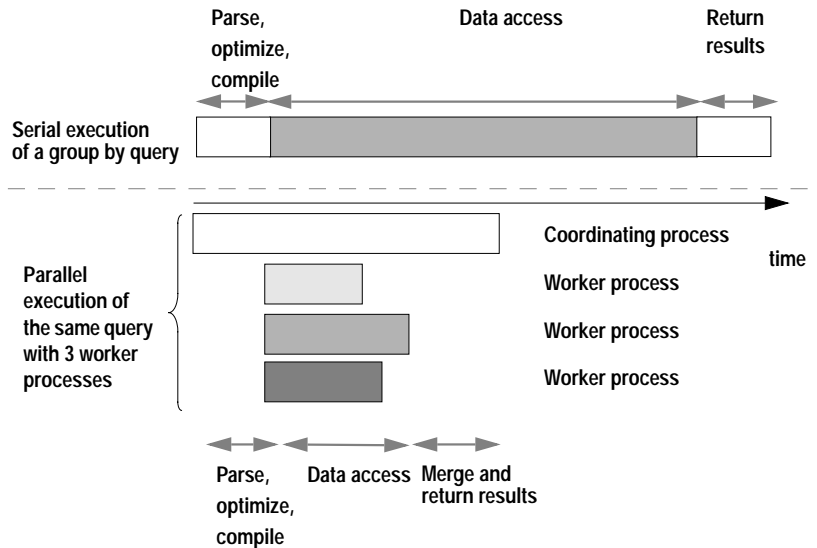


Figure 13-2: Relative execution times for serial and parallel query execution

The total amount of work performed by the query running in parallel is more than the amount of work performed by the query running in serial, but the response time is shorter.

Returning Results from Parallel Queries

Results from parallel queries are returned through one of three merge strategies, or as the final step in a sort.

Parallel queries that do not have a final sort step use one of these merge types:

- Queries that contain a vector (grouped) aggregate use worktables to store temporary results; the coordinating process merges these into one worktable and returns results to the client.
- Queries that contain a scalar (ungrouped) aggregate use internal variables, and the coordinating process performs the final computations to return the results to the client.
- Queries that do not contain aggregates or use clauses that do not require a sort can return results to the client as the tables are being scanned. Each worker process stores results in a result buffer and

uses address locks to coordinate transferring the results to the network buffers for the task.

More than one merge type can be used when queries require several steps or multiple worktables. See “showplan Messages for Parallel Queries” on page 9-42 for more information on merge messages.

For parallel queries that include an *order by* clause, *distinct*, or *union*, results are stored in a worktable in *tempdb*, then sorted. If the sort can benefit from parallel sorting, a parallel sort is used, and results are returned to the client during the final merge step performed by the sort. For more information on how parallel sorts are performed, see Chapter 15, “Parallel Sorting.”

► **Note**

Since parallel queries use multiple processes to scan data pages, queries that do not use aggregates and do not include a final sort set may return results in different order than serial queries and may return different results for queries with *set rowcount* in effect and for queries that select into a local variable. For details and solutions, see “When Parallel Query Results Can Differ” on page 13-24.

Types of Parallel Data Access

Adaptive Server accesses data in parallel in different ways, depending on the settings of the configuration parameters, whether tables are partitioned, and the availability of indexes. The optimizer uses this information to choose the query plan. The optimizer may choose a mix of serial and parallel methods in queries that involve multiple tables or multiple steps. Parallel methods include:

- Hash-based table scans
- Hash-based nonclustered index scans
- Partition-based scans, either full table scans or scans positioned with a clustered index

showplan reports the type of parallel access method, as well as the number of worker processes, used in the query. See “showplan Messages for Parallel Queries” on page 9-42 for examples.

Figure 13-3 shows a scan on a table executed in serial by a single task. The task follows the table's page chain to read each page, stopping to perform physical I/O when needed pages are not in the cache.

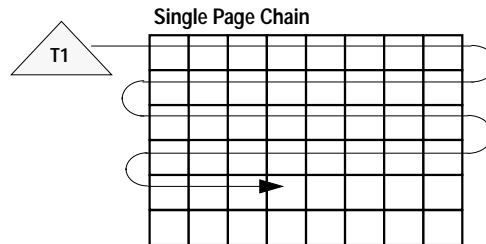


Figure 13-3: A serial task scans data pages

Hash-Based Table Scans

Figure 13-4 shows how three worker processes divide the work of accessing a table's data pages during a hash-based table scan. Each worker process performs a logical I/O on every page in a hash-based table scan, but each process examines rows on only one-third of the pages, as indicated by the differently shaded pages. Hash-based table scans are used only for the outer query in a join.

With only one engine, the query still benefits from parallel access because one worker process can execute while others wait for I/O. If there were multiple engines, some of the worker processes could be running simultaneously.

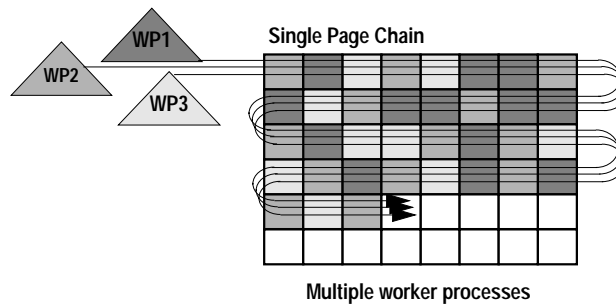


Figure 13-4: Worker processes scan an unpartitioned table

Hash-based table scans increase the logical I/O for the scan, since each worker process must access each page to hash on the page ID.

Partition-Based Scans

Figure 13-5 shows how a query scans a table that has three partitions on three physical disks. With a single engine, this query can benefit from parallel processing because one worker process can execute while others sleep on I/O or wait for locks held by other processes to be released. If multiple engines are available, the worker processes can run simultaneously. This configuration can yield high parallel performance by providing I/O parallelism.

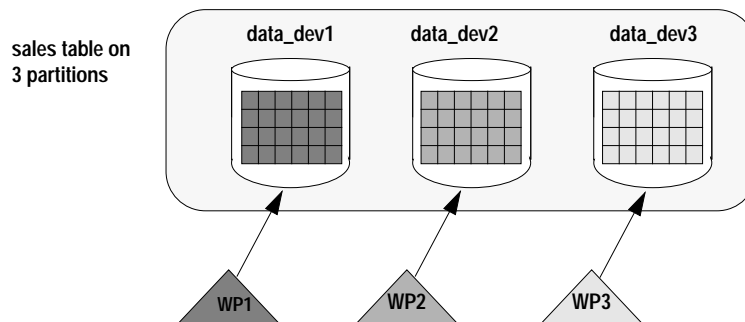


Figure 13-5: Multiple worker processes access multiple partitions

Hash-Based Nonclustered Index Scans

Figure 13-6 shows a hash-based nonclustered index scan. Each worker process navigates higher levels of the index and reads the leaf-level pages of the index. Each worker process then hashes on either the data page ID or the key value to determine which data

pages or data rows to process. Reading every leaf page produces negligible overhead.

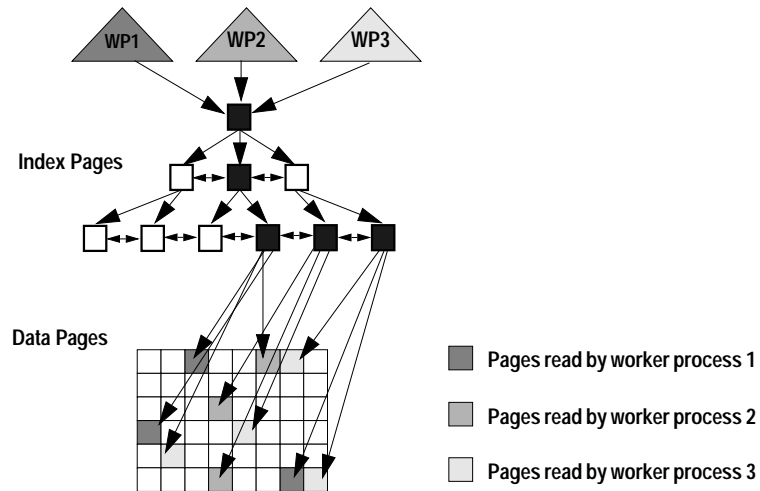


Figure 13-6: Hash-based, nonclustered index scan

Parallel Processing for Two Tables in a Join

Figure 13-7 shows a join query performing a partition-based scan on a table with three partitions, and a hash-based, nonclustered index scan, with two worker processes on the second table. When parallel access methods are used on more than one table in a join, the total number of worker processes required is the product of worker process for each scan. In this case, six workers perform the query, with each worker process scanning both tables. Two worker processes scan each partition in the first table, and all six worker processes navigate the index tree for the second table and scan the leaf pages. Each worker process accesses the data pages that correspond to its hash value.

The optimizer chooses a parallel plan for a table only when a scan returns 20 pages or more. These types of join queries require 20 or

more matches on the join key for the inner table in order for the inner scan to be optimized in parallel.

Table1:
Partitioned table
on 3 devices

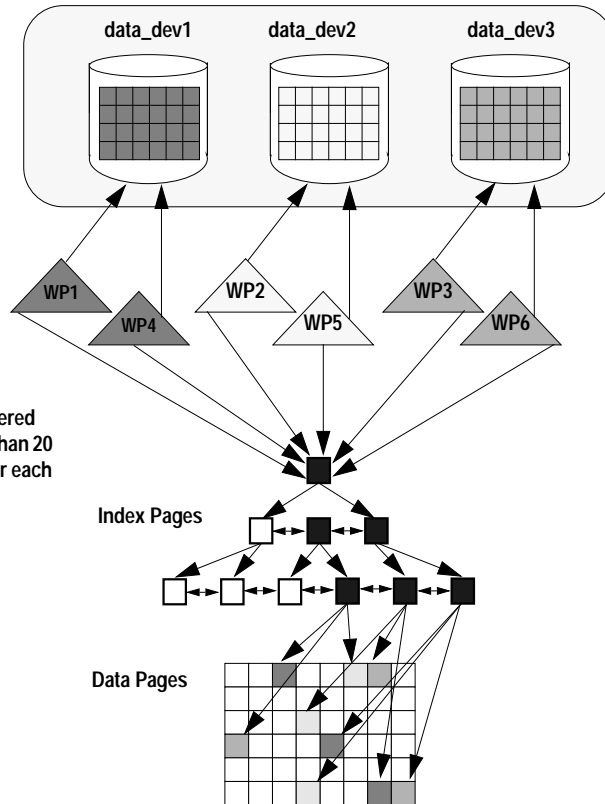


Table2: Nonclustered
index with more than 20
matching rows for each
join key

Figure 13-7: Join query using different parallel access methods on each table

showplan Messages for Parallel Queries

showplan prints the degree of parallelism each time a table is accessed in parallel. The following example shows the messages for each table in the join in Figure 13-7:

```
Executed in parallel with a 2-way hash scan.
```

```
Executed in parallel with a 3-way partition scan.
```

showplan also prints a message showing the total number of worker processes used, the degree of parallelism for the entire query. For the query shown in Figure 13-7, it reports:

Executed in parallel by coordinating process and 6 worker processes.

See “showplan Messages for Parallel Queries” on page 9-42 for more information and Chapter 14, “Parallel Query Optimization,” for additional examples.

Controlling the Degree of Parallelism

A parallel query’s **degree of parallelism** is the maximum number of worker processes used to execute the query. This number depends on several factors, including the values to which you set the parallel configuration parameters or session-level limits, (see Table 13-1 and Table 13-2), the number of partitions on a table (for partition-based scans), the level of parallelism suggested by the optimizer, and the number of worker processes that are available at the time the query executes.

You can establish limits on the degree of parallelism:

- Server-wide—using the `sp_configure` system procedure with parameters shown in Table 13-1. Only a System Administrator can use `sp_configure`.
- For a session—the `set` command with the parameters shown in Table 13-2. All users can issue the `set` command, and it can be included in stored procedures.
- In a select query—using the `parallel` clause, as shown in “Controlling Parallelism for a Query” on page 13-14.

Configuration Parameters for Controlling Parallelism

The configuration parameters that give you control over the degree of parallelism server-wide are shown in Table 13-1.

Table 13-1: Configuration parameters for parallel execution

Parameter	Explanation	Comment
number of worker processes	The maximum number of worker processes available for all parallel queries. Each worker process requires approximately as much memory as a user connection.	Restart of server required
max parallel degree	The number of worker processes that can be used by a single query. It must be equal to or less than number of worker processes and equal to or greater than max scan parallel degree .	Dynamic, no restart required
max scan parallel degree	The maximum number of worker processes that can be used for a hash scan. It must be equal to or less than number of worker processes and max parallel degree .	Dynamic, no restart required

You must restart Adaptive Server after changing **number of worker processes**. Each worker process uses approximately the same amount of memory as a user connection. Configuring **number of worker processes** affects the size of the data and procedure cache, so you may want to change the value of **total memory** also. See “Parallel Queries” on page 11-81 in the *System Administration Guide*.

max parallel degree and **max scan parallel degree** are dynamic options, so a System Administrator can change them without restarting the server. When you change either of these parameters, all query plans in cache are invalidated, so the next execution of any stored procedure recompiles the plan and uses the new values.

How Limits Apply to Query Plans

When queries are optimized, this is how the configuration parameters affect query plans:

- **max parallel degree** limits:
 - The number of worker processes for a partition-based scan
 - The total combined number of worker processes for join queries, where parallel accesses methods are used on more than one table

- The number of worker processes that can be used by parallel sort operations
- `max scan parallel degree` limits the number of worker processes for hash-based table scans and nonclustered index scans

How the Limits Work In Combination

You might configure `number of worker processes` to 50 to allow multiple parallel queries to operate at the same time. If the table with the largest number of partitions has 10 partitions, you might set `max parallel degree` to 10, limiting all select queries to a maximum of 10 worker processes. Since hash-based scans operate best with 2–3 worker processes, `max scan parallel degree` could be set to 3.

For a single-table query, or a join involving serial access on other tables, some of the parallel possibilities allowed by these values are:

- Parallel partition scans on any tables with 2–10 partitions
- Hash-based table scans with up to 3 worker processes
- Hash-based nonclustered index scans on tables with nonclustered indexes, with up to 3 worker processes

For joins where parallel methods are used on more than one table, some possible parallel choices are:

- Joins using a hash-based scan on one table and partitioned-based scans on tables with 2 or 3 partitions
- Joins using partitioned based scans on both tables. For example:
 - A parallel degree of 3 for a partitioned table multiplied by `max scan parallel degree` of 3 for a hash-based scan requires 9 worker processes.
 - A table with 2 partitions and a table with 5 partitions requires 10 worker processes for partition-based scans on both tables.
 - Tables with 4–10 partitions can be involved in a join, with one or more tables accessed in serial.

For fast performance, while creating a clustered index on a table with 10 partitions, the setting of 50 for `number of worker processes` allows you to set `max parallel degree` to 20 for the `create index` command. For more information on configuring worker processes for sorting, see “Worker Process Requirements During Parallel Sorts” on page 15-6.

Examples of Setting Parallel Configuration Parameters

The following command sets number of worker processes:

```
sp_configure "number of worker processes", 50
```

After a restart of the server, these commands set the other configuration parameters:

```
sp_configure "max parallel degree", 10
sp_configure "max scan parallel degree", 3
```

To display the current settings for these parameters, use this command:

```
sp_configure "Parallel Query"
```

Using *set* Options to Control Parallelism for a Session

Two set options let you restrict the degree of parallelism on a session basis or in stored procedures or triggers. These options are useful for tuning experiments with parallel queries and can also be used to restrict noncritical queries to run in serial, so that worker processes remain available for other tasks. The set options are summarized in Table 13-2.

Table 13-2: set options for parallel execution tuning

Parameter	Function
<code>parallel_degree</code>	Sets the maximum number of worker processes for a query in a session, stored procedure, or trigger. Overrides the <code>max parallel degree</code> configuration parameter, but must be less than or equal to the value of <code>max parallel degree</code> .
<code>scan_parallel_degree</code>	Sets the maximum number of worker processes for a hash-based scan during a specific session, stored procedure, or trigger. Overrides the <code>max scan parallel degree</code> configuration parameter but must be less than or equal to the value of <code>max scan parallel degree</code> .

If you specify a value that is too large for either option, the value of the corresponding configuration parameter is used, and a message reports the value in effect. While `set parallel_degree` or `set scan_parallel_degree` is in effect during a session, the plans for any stored procedures that you execute are not placed in the procedure

cache. Procedures run with these options in effect may produce suboptimal plans.

set Command Examples

This example restricts all queries started in the current session to 5 worker processes:

```
set parallel_degree 5
```

While this command is in effect, any query on a table with more than 5 partitions cannot be run in parallel.

To remove the session limit, use:

```
set parallel_degree 0
or
set scan_parallel_degree 0
```

To run subsequent queries in serial mode, use:

```
set parallel_degree 1
or
set scan_parallel_degree 1
```

Controlling Parallelism for a Query

The `parallel` extension to the `from` clause of a `select` command allows users to suggest the number of worker processes used in a `select` statement. The degree of parallelism that you specify cannot be more than the value set with `sp_configure` or the session limit controlled by a `set` command. If you specify a higher value, the specification is ignored, and the optimizer uses the `set` or `sp_configure` limit.

The syntax for the `select` statement is:

```
select ...
  from tablename [( [index index_name]
                   [parallel [degree_of_parallelism | 1 ]]
                   [prefetch size] [lru|mru] ) ] ,
  tablename [( [index index_name]
               [parallel [degree_of_parallelism | 1]]
               [prefetch size] [lru|mru] ) ] ...
```

Query Level *parallel* Clause Examples

To specify the degree of parallelism for a single query, include `parallel` after the table name. This example executes in serial:

```
select * from huge_table (parallel 1)
```

This example specifies the index to use in the query, and sets the degree of parallelism to 2:

```
select * from huge_table (index ncix parallel 2)
```

To force a table scan, use the table name instead of the index name.

See “Suggesting a Degree of Parallelism for a Query” on page 10-15 for more information.

Worker Process Availability and Query Execution

At run time, if the number of worker processes specified in the query plan is not available, Adaptive Server creates an adjusted query plan to execute the query using fewer worker processes. This is called a **run-time adjustment**, and it can result in serial execution of the query.

A run-time adjustment now and then probably indicates an occasional, momentary bottleneck. Frequent run-time adjustments indicate that the system may not be configured with enough worker processes for the workload. See “Run-Time Adjustment of Worker Processes” on page 14-29 for more information. You can also use the set `process_limit_action` option to control whether a query or stored procedure should silently use an adjusted plan, whether it should warn the user, or whether the command should fail if it cannot use the optimal number of worker processes. See “Using set `process_limit_action`” on page 14-31 for more information.

Run-time adjustments are transparent to end users, except:

- A query that normally runs in parallel may perform very slowly in serial
- If set `process_limit_action` is in effect, they may get a warning, or the query may be aborted, depending on the setting

Other Configuration Parameters for Parallel Processing

Two additional configuration parameters for parallel query processing are:

- **number of sort buffers**—configures the maximum number of buffers that parallel sort operations can use from the data cache. See “Caches, Sort Buffers, and Parallel Sorts” on page 15-10 for information.

- **memory per worker process**—establishes a pool of memory that all worker processes use for messaging during query processing. The default value, 1024 bytes per worker process, provides ample space in almost all cases, so this value should not need to be reset. See “Worker Process Management” on page 24-16 for information on monitoring and tuning this value.

Commands for Working with Partitioned Tables

Detailed steps for partitioning tables, placing them on specific devices, and loading data with parallel bulk copy are included in Chapter 17, “Controlling Physical Data Placement.” The commands and tasks for creating, managing, and maintaining partitioned tables are:

- **alter database**—to make devices available to the database
- **sp_addsegment**—to create a segment on a device; **sp_extendsegment** to extend the segment over additional devices, and **sp_dropsegment** to drop the log and system segments from data devices
- **create table...on *segment_name***—to create a table on a segment
- **alter table...partition** and **alter table...unpartition**—to add or remove partitioning from a table
- **create clustered index**—to distribute the data evenly across the table’s partitions
- **bcp (bulk copy)**—with the partition number added after the table name, to copy data into specific table partitions
- **sp_helppartition**—to display the number of partitions and the distribution of data in partitions, and **sp_helpsegment** to check the space used on each device in a segment and on the segment as a whole

Figure 13-8 shows a possible scenario for creating a new partitioned table.

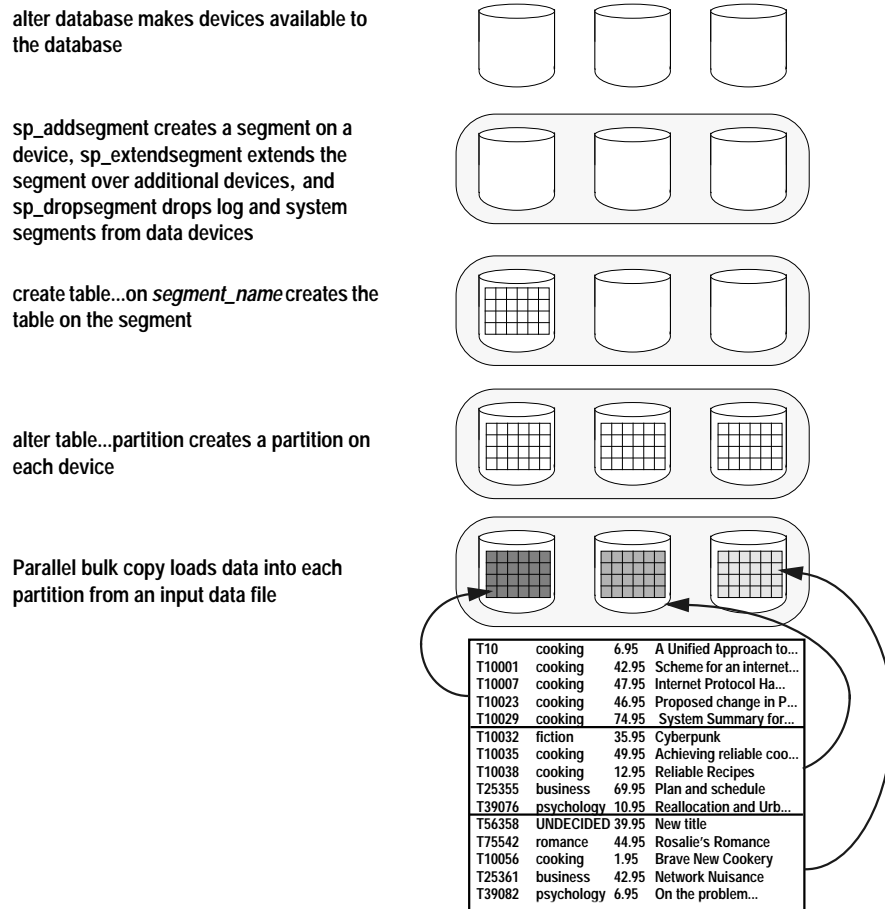


Figure 13-8: Steps for creating and loading a new partitioned table

Balancing Resources and Performance

Maximum parallel performance requires multiple CPUs and multiple I/O devices to achieve I/O parallelism. As with most performance configuration, parallel systems reach a point of diminishing returns, and a later point where additional resources do not yield performance improvement.

You need to determine whether queries are CPU intensive or I/O intensive and when your performance is blocked by CPU saturation

or I/O bottlenecks. If CPU utilization is low, spreading a table across more devices and using more worker processes increases CPU utilization and provides improved response time. Conversely, if CPU utilization is extremely high, but the I/O system is not saturated, increasing the number of CPUs can provide performance improvement.

CPU Resources

Without an adequate number of engines (CPU resources), tasks and worker processes must wait for access to Adaptive Server engines, and response time can be slow. Many factors determine the number of engines needed by the system, such as whether the query is CPU intensive or I/O intensive, or, at different times, both:

- Worker processes tend to spend time waiting for disk I/O and other system resources while other tasks are active on the CPU.
- Queries that perform sorts and aggregates tend to be more CPU intensive.
- Execution classes and engine affinity bindings on parallel CPU-intensive queries can have complex effects on the system. If there are not enough CPUs, performance for all queries, both serial and parallel, can be degraded. See Chapter 22, “Distributing Engine Resources Between Tasks,” for more information.

Disk Resources and I/O

In most cases, configuring the physical layout of tables and indexes on devices is the key to parallel performance. Spreading partitions across different disks and controllers can improve performance during partition-based scanning if all of the following conditions are true:

- Data is distributed over different disks,
- Those disks are distributed over different controllers, and
- There are enough worker processes available at run time to allocate one worker process for each partition.

Tuning Example: CPU and I/O Saturation

One experiment on a CPU-bound query found near-linear scaling in performance by adding CPUs until the I/O subsystem became

saturated. At that point, additional CPU resources did not improve performance. The query performs a table scan on an 800MB table with 30 partitions, using 16K I/O. Table 13-3 shows the CPU scaling.

Table 13-3: Scaling of engines and worker processes

Engines	Elapsed Time, (in Seconds)	CPU Utilization	I/O Saturation	Throughput per Device, per Second
1	207	100%	Not saturated	.13MB
2	100	98.7%	Not saturated	.27MB
4	50	98%	Not saturated	.53MB
8	27	93%	100% saturated	.99MB

Guidelines for Parallel Query Configuration

Parallel processing places very different demands on system resources than running the same queries in serial. Two components in planning for parallel processing are:

- A good understanding of the capabilities of the underlying hardware (especially disk drives and controllers) in use on your system
- A set of performance goals for queries you plan to run in parallel

Hardware Guidelines

Some guidelines for hardware configuration and disk I/O speeds are:

- Each Adaptive Server engine can support about five worker processes before saturating on CPU utilization for CPU-intensive queries. If CPU is not saturated at this ratio, and you want to improve parallel query performance, increase the ratio of worker processes to engines until I/O bandwidth becomes a bottleneck.
- For sequential scans, such as table scans using 16K I/O, it may be possible to achieve 1.6MB per second, per device, that is, 100 16K I/Os, or 800 pages per second, per device.
- For queries doing random access, such as nonclustered index access, the figure is approximately 50 2K I/Os, or 50 pages per second, per device.

- One I/O controller can sustain a transfer rate of up to 10–18MB per second. This means that one SCSI I/O controller can support up to 6–10 devices performing sequential scans. Some high-end disk controllers can support more throughput. Check your hardware specifications, and use sustained rates, rather than peak rates, for your calculations.
- RAID disk arrays vary widely in performance characteristics, depending on the RAID level, the number of devices in the stripe set, and specific features, such as caching. RAID devices may provide better or worse throughput for parallelism than the same number of physical disks without striping. In most cases, start your parallel query tuning efforts by setting the number of partitions for tables on these devices to the number of disks in the array.

Working with Your Performance Goals and Hardware Guidelines

The following examples use the hardware guidelines and the example reported in Table 13-3 to provide some examples of using parallelism to meet performance goals:

- The number of partitions for a table should be less than or equal to the number of devices. For the experiment showing scaling of engines and worker processes shown in Table 13-3, there were 30 devices available, so 30 partitions were used. Performance is optimal when each partition is placed on a separate physical device.
- Determine the number of partitions based on the I/O throughput you want to achieve. If you know your disks and controllers can sustain 1MB per second, per device, and you want a table scan on an 800MB table to complete in 30 seconds, you need to achieve approximately 27MB per second total throughput, so you would need at least 27 devices with one partition per device, and at least 27 worker processes, one for each partition. These figures are very close to the I/O rates in the example in Table 13-3.
- Estimate the number of CPUs, based on the number of partitions, and then determine the optimum number by tracking both CPU utilization and I/O saturation. The example shown in Table 13-3 had 30 partitions available. Following the suggestions in the hardware guidelines of 1 CPU for each 5 devices suggests using 6 engines for CPU-intensive queries. At that level, I/O was not saturated, so adding more engines improved response time.

Examples of Parallel Query Tuning

The following examples use the I/O capabilities described in “Hardware Guidelines” on page 13-18.

Improving the Performance of a Table Scan

This example shows how a table might be partitioned to meet performance goals. Queries that scan whole tables and return a limited number of rows are good candidates for parallel performance. An example is this query containing `group by`:

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

Here are the performance statistics and tuning goals:

Table size	48,000 pages
Access method	Table scan, 16K I/O
Serial response time	60 seconds
Target performance	6 seconds

The steps for configuring for parallel operation are:

- Create 10 partitions for the table, and evenly distribute the data across the partitions
- Set the number of worker processes and max parallel degree configuration parameters to at least 10
- Check that the table uses a cache configured for 16K I/O

In serial execution, 48,000 pages were scanned in 60 seconds using 16K I/O. In parallel execution, each process scans 1 partition, approximately 4,800 pages, in about 6 seconds, again using 16K I/O.

Improving the Performance of a Nonclustered Index Scan

The following example shows how performance of a query using a nonclustered index scan can be improved by configuring for a hash-based scan. The performance statistics and tuning goals are:

Data pages accessed	1500
Access method	Nonclustered index, 2K I/O
Serial response time	30 seconds
Target performance	6 seconds

The steps for configuring for parallel operation are:

- Set `max scan parallel degree` configuration parameters to 5 to use 5 worker processes in the hash-based scan
- Set `number of worker processes` and `max parallel degree` to at least 5

In parallel execution, each worker process scans 300 pages in 6 seconds.

Guidelines for Partitioning and Parallel Degree

Here are some additional guidelines to consider when you are moving from serial query execution to parallel execution or considering additional partitioning or additional worker processes for a system already running parallel queries:

- If the cache hit ratio for a table is more than 90 percent, partitioning the table will not greatly improve performance. Since most of the needed pages are in cache, there is no benefit from the physical I/O parallelism.
- If CPU utilization is more than 80 percent, and a high percentage of the queries in your system make use of parallel queries, increasing the degree of parallelism may cause CPU saturation. This guideline also applies to moving from all-serial query processing to parallel query processing, where a large number of queries are expected to make use of parallelism. Consider adding more engines, or start with a low degree of parallelism.
- If CPU utilization is high, and a few users run large DSS queries while most users execute OLTP queries that do not operate in parallel, enabling or increasing parallelism can improve response time for the DSS queries. However, if response time for OLTP

queries is critical, start with a low degree of parallelism, or make small changes to the existing degree of parallelism.

- If CPU utilization is low, move incrementally toward higher degrees of parallelism. On a system with two CPUs, and an average CPU utilization of 60 percent, doubling the number of worker processes would saturate the CPUs.
- If I/O for the devices is well below saturation, you may be able to improve performance for some queries by breaking the one-partition-per-device guideline. Except for RAID devices, always use a multiple of the number of logical devices in a segment for partitioning; that is, for a table on a segment with four devices, you can use eight partitions. Doubling the number of partitions per device may cause extra disk-head movement and reduce I/O parallelism. Note that creating an index on any partitioned table that has more partitions than devices prints a warning message that you can ignore in this case.

Experimenting with Data Subsets

Parallel query processing can provide the greatest performance gains on your largest tables and most I/O intensive queries. Experimenting with different physical layouts on huge tables, however, is extremely time-consuming. Here are some suggestions for working with smaller subsets of data:

- For initial exploration to determine the types of query plans that would be chosen by the optimizer, experiment with a proportional subset of your data. For example, if you have a 50-million row table that joins to a 5-million row table, you might choose to work with just one-tenth of the data, using 5 million and 500,000 rows. Be sure to select subsets of the tables that provide valid joins. Pay attention to join selectivity—if the join on the table would run in parallel because it would return 20 rows for a scan, be sure your subset reflects this join selectivity.
- The optimizer does not take underlying physical devices into account, but only the partitioning on the tables. During exploratory tuning work, distributing your data on separate physical devices will give you more accurate predictions about the probable characteristics of your production system using the full tables. You can partition tables that reside on a single device and ignore any warning messages during the early stages of your planning work, such as testing configuration parameters, table

partitioning and checking your query optimization. Of course, this does not provide accurate I/O statistics.

Working with subsets of data can help determine parallel query plans and the degree of parallelism for tables. One difference is that with smaller tables, sorts are performed in serial that would be performed in parallel on larger tables.

System Level Impacts

In addition to other impacts described throughout this chapter, here are some concerns to be aware of when adding parallelism to mixed DSS and OLTP environments. Your goal should be improved performance of DSS through parallelism, without adverse effects on the performance of OLTP applications.

Locking Issues

- Look out for lock contention:
 - Parallel queries will be slower than queries benchmarked without contention. If the scans find many pages with exclusive locks due to update transactions, performance can change.
 - If parallel queries return a large number of rows using network buffer merges, there is likely to be high contention for the network buffer. These queries hold shared locks on data pages during the scans and can cause data modifications to wait for the shared locks to be released. You may need to restrict these types of queries to serial operation.
 - If your applications experience deadlocks when DSS queries are running in serial, you may see an increase in deadlocks when you run these queries in parallel. The transaction that is rolled back in these deadlocks is likely to be the OLTP queries, because the rollback decision for deadlocks is based on the accumulated CPU time of the processes involved. See “Deadlocks and Concurrency” on page 5-26 for more information on deadlocks.

Device Issues

Configuring multiple devices for *tempdb* should improve performance for parallel queries that require worktables, including

those that perform sorts and aggregates and those that use the reformatting strategy.

Procedure Cache Effects

Parallel query plans are slightly larger than serial query plans because they contain extra instructions on the partition or pages that the worker processes need to access.

During ad hoc queries, each worker process needs a copy of the query plan. Space from the procedure cache is used to hold these plans in memory, and is available to the procedure cache again when the ad hoc query completes.

Stored procedures in cache are invalidated when you change the parallel configuration parameters `max parallel degree` and `max scan parallel degree`. The next time a query is run, the query is read from disk and recompiled.

When Parallel Query Results Can Differ

When a query does not include vector or scalar aggregates or does not require a final sorting step, a parallel query might return results in a different order from the same query run in serial, and subsequent executions of the same query in parallel might return results in different order each time.

Results from serial and parallel queries that include vector or scalar aggregates, or require a final sort step, are returned after all of the results from worktables are merged or sorted in the final query processing step. Without query clauses that require this final step, parallel queries send results to the client using a network buffer merge, that is, each worker process sends results to the network buffer as it retrieves the data that satisfies the queries.

The relative speed of the different worker processes leads to the differences in result set ordering. Each parallel scan behaves a little differently, due to pages already in cache, lock contention, and so forth. Parallel queries always return the same **set** of results, just not in the same **order**. If you need a dependable ordering of results, you must use `order by` or run the query in serial mode.

In addition, due to the pacing effects of multiple worker processes reading data pages, two types of queries may return different results accessing the same data when an aggregate or a final sort is not done:

- Queries that use `set rowcount`
- Queries that select a column into a local variable without sufficiently restrictive query clauses

Queries That Use `set rowcount`

The `set rowcount` option stops processing after a certain number of rows are returned to the client. With serial processing, the results are consistent in repeated executions. In serial mode, the same rows are returned in the same order for a given `rowcount` value, because a single process reads the data pages in the same order every time.

With parallel queries, the order of the results and the set of rows returned can differ, due to the fact that worker processes may access pages sooner or later than other processes. When `set rowcount` is in effect, each row is written to the network buffer as it is found and the buffer is sent to the client when it is full, until the required number of rows have been returned. To get consistent results, you must either use a clause that performs a final sort step or run the query in serial mode.

Queries That Set Local Variables

This query sets the value of a local variable in a `select` statement:

```
select @tid = title_id from titles
       where type = "business"
```

The `where` clause in the query matches multiple rows in the `titles` table, so the local variable is always set to the value from the last matching row returned by the query. The value is always the same in serial processing, but for parallel query processing, the results depend on which worker process finishes last. To achieve a consistent result, use a clause that performs a final sort step, execute the query in serial mode, or add clauses so that the query arguments select only single rows.

Achieving Consistent Results

To achieve consistent results for the types of queries discussed in this section, you can either add a clause to enforce a final sort or you can run the queries in serial mode. The query clauses that provide a final sort are:

- **order by**
- **distinct**, except for uses of **distinct** within an aggregate, such as **avg(distinct price)**
- **union**, but not **union all**

To run queries in serial mode, you can:

- Use set **parallel_degree 1** to limit the session to serial operation
- Include the **(parallel 1)** clause after each table listed in the **from** clause of the query

14

Parallel Query Optimization

This chapter describes the basic strategies that Adaptive Server uses to perform **parallel** queries and explains how the optimizer applies those strategies to different queries. It is not necessary to understand these concepts in order to benefit from parallel queries; parallel query optimization is an automatic process, and the optimized query plans created by Adaptive Server generally yield the best performance for a particular query.

However, knowing the internal workings of a parallel query can help you understand why queries are sometimes executed in serial, or with fewer worker processes than you expected. Knowing why these events occur can help you make changes elsewhere in your system to ensure that certain queries are executed in parallel and with the desired number of processes.

This chapter contains the following sections:

- What Is Parallel Query Optimization? 14-1
- When Is Parallel Query Optimization Performed? 14-3
- Overhead Cost of Parallel Queries 14-3
- Parallel Access Methods 14-4
- Summary of Parallel Access Methods 14-11
- Degree of Parallelism for Parallel Queries 14-13
- Parallel Strategies for Joins 14-17
- Parallel Query Examples 14-21
- Run-Time Adjustment of Worker Processes 14-30
- Diagnosing Parallel Performance Problems 14-37
- Resource Limits for Parallel Queries 14-39

What Is Parallel Query Optimization?

Parallel query optimization is the process of analyzing a query and choosing the best combination of **parallel** and **serial** access methods to yield the fastest response time for the query. Parallel query optimization is an extension of the serial optimization strategies discussed in Chapter 8, “Understanding the Query Optimizer.” As

with serial query optimization, parallel optimization involves three basic steps:

1. Analyzing the query to identify key search arguments, and associate those arguments with the correct tables.
2. Identifying the best indexes and search clauses, based on cost estimates.
3. Performing an exhaustive search of the join ordering and indexes, and analyzing the cost of different access methods for each combination.

Parallel query optimization differs from standard optimization mainly in step 3 above, where parallel access methods are considered in addition to serial methods. The optimizer may choose between any combination of serial and parallel access methods to create the fastest query plan.

Optimizing for Response Time vs. Total Work

Serial query optimization, introduced in Chapter 8, “Understanding the Query Optimizer”, compares the cost of different query plans and selects the plan that is the least costly to execute. Since only one process executes the query, choosing the least costly plan yields the fastest response time **and** requires the least amount of total work from the server.

The goal of executing queries in parallel is to get the fastest response time, even if it involves more total work from the server. During parallel query optimization, the optimizer uses cost-based comparisons similar to those used in serial optimization to select a final query plan.

However, since multiple worker processes execute the query, the parallel query plan invariably requires more total work from Adaptive Server. Multiple worker processes, engines, and partitions that improve the speed of a query require additional costs in overhead, CPU utilization, and disk access. In other words, whereas serial query optimization improves performance by minimizing the use of server resources, parallel query optimization improves performance for individual queries by fully utilizing available resources in order to get the fastest response time.

When Is Parallel Query Optimization Performed?

The Adaptive Server optimizer uses parallel optimization strategies only when a given query is “eligible” for parallel execution. This means that the Adaptive Server and the current session are properly configured for parallelism, as described in “Controlling the Degree of Parallelism” on page 13-11.

If both the Adaptive Server and the current session are configured for parallel queries, then all queries within the session are eligible for parallel query optimization. Individual queries can also attempt to enforce parallel query optimization by using the optimizer hint `parallel N` for parallel or `parallel 1` for serial.

If the Adaptive Server or the current session is not configured for parallel queries, or if a given query uses optimizer hints to enforce serial execution, then the optimizer uses only the techniques described in Chapter 8, “Understanding the Query Optimizer”; the parallel access methods described in this chapter are not considered.

► *Note*

Adaptive Server does not execute parallel queries against system tables.

Overhead Cost of Parallel Queries

Parallel queries incur more overhead costs to perform such internal tasks as:

- Allocating and initializing worker processes
- Coordinating worker processes as they execute a query plan
- Deallocating worker processes after the query is completed

To avoid applying these overhead costs to OLTP-based queries, the optimizer “disqualifies” tables from using parallel access methods when a scan would access fewer than 20 data pages in those tables. This restriction applies whether or not an index is used to access a table’s data. When Adaptive Server must scan fewer than 20 pages of a table’s data, the optimizer considers only serial table and index scans and does not apply parallel optimization strategies.

Factors That Are Not Considered

When computing the cost of a parallel access method, the optimizer **does not** consider factors such as the number of engines available, the ratio of engines to CPUs, and whether or not a table's partitions reside on dedicated physical devices and controllers. Each of these factors can significantly affect the performance of a query. It is up to the System Administrator to ensure that these resources are configured in the best possible way for the Adaptive Server system as a whole. See "Parallel Queries" on page 11-81 in the *System Administration Guide* for information on configuring Adaptive Server. See "Commands for Partitioning Tables" on page 17-22 for information on partitioning your data to best facilitate parallel queries.

Parallel Access Methods

The following sections describe parallel **access methods** and other strategies that the optimizer considers when optimizing parallel queries. An access method is a mechanism used by Adaptive Server to scan data from a table to produce results for a query. Two common serial access methods are the **table scan** and the **clustered index scan**. The parallel access methods described in this section are considered **in addition to** the serial methods described in "Optimizer Strategies" on page 8-8.

Parallel access methods fall into two general categories: partition-based methods and hash-based methods.

- **Partition-based** methods use two or more worker processes to access separate partitions of a table. Partition-based methods yield the fastest response times because they can distribute the work in accessing a table over both CPUs and physical disks. At the CPU level, worker processes can be queued to separate engines to increase processing performance. At the physical disk level, worker processes can perform I/O independently of one another, if the table's partitions are distributed over separate physical devices and controllers.
- **Hash-based** methods provide parallel access to partitioned tables, unpartitioned tables, and nonclustered index pages. Hash-based strategies employ multiple worker processes to work on a single chain of data pages or a set of index pages. I/O is not distributed over physical devices or controllers, but worker

processes can still be queued to multiple engines to distribute processing and improve response times.

Parallel Partition Scan

In a parallel partition scan, multiple worker processes completely scan each partition in a partitioned table. One worker process is assigned to each partition, and each process reads pages from the first page in the partition to the last. Figure 14-1 illustrates a parallel partition scan.

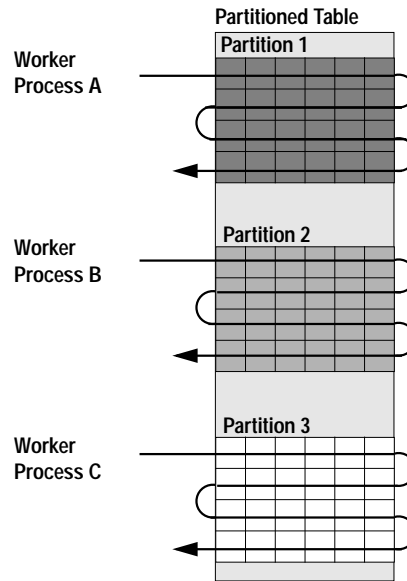


Figure 14-1: Parallel partition scan

The parallel partition scan operates faster than a serial table scan. The work is divided over several worker processes that can execute simultaneously on different engines. Some worker processes can be executing during the time that others sleep on I/O or other system resources. If the table partitions reside on separate physical devices, I/O parallelism is also possible.

Requirements for Consideration

The optimizer considers the parallel partition scan only for partitioned tables in a query. The table's data must not be skewed in

relation to the number of partitions, or the optimizer disqualifies partition-based access methods from consideration. Table data is considered skewed when the size of the largest partition is two or more times the average partition size.

Finally, the query must access at least 20 data pages before the optimizer considers any parallel access methods.

Cost Model

The Adaptive Server optimizer computes the cost of a parallel table partition scan as the largest number of page I/Os performed by any one worker process in the scan. In other words, the cost of this access method is equal to the number of pages in the largest partition of the table.

For example, if a table with 3 partitions has 200 pages in its first partition, 300 pages in its second, and 500 pages in its last partition, the cost of performing a partition scan on that table is 500 page I/Os. In contrast, the cost of a serial scan of this table is 1000 page I/Os.

Parallel Hash-Based Table Scan

In a hash-based table scan, multiple worker processes scan a single chain of data pages in a table at the same time. All worker processes traverse the page chain and apply an internal hash function to each page ID. The hash function determines which worker process reads the rows in the current page. The hash function ensures that only one worker process scans the rows on any given page of the table. Figure 14-2 illustrates the hash-based table scan.

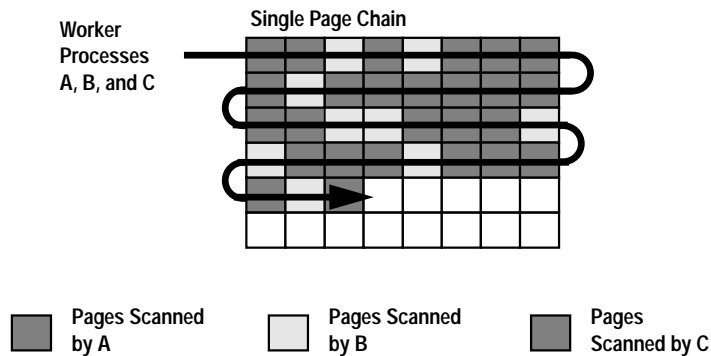


Figure 14-2: Parallel hash-based table scan

The hash-based scan provides a way to distribute the processing of a single chain of data pages over multiple engines. The optimizer may use this access method for the outer table of a join query to process a join condition in parallel.

Requirements for Consideration

The optimizer considers the hash-based table scan only for heap tables, and only for outer tables in a join query—it does not consider this access method for tables with clustered indexes or single-table queries. Hash-based scans can be used on either unpartitioned or partitioned tables. The query must access at least 20 data pages of the table before the optimizer considers any parallel access methods.

Cost Model

The Adaptive Server optimizer computes the cost of a hash-based table scan as the total number of page I/Os required to scan the table. This is the same cost applied to a serial table scan.

Parallel Clustered Index Partition Scan

A clustered index partition scan uses multiple worker processes to scan data pages in a partitioned table when the clustered index key matches a search argument (SARG). At execution time, Adaptive Server assigns one worker process to each partition in the table that must be scanned. Each worker process then accesses data pages in the partition, using one of two methods, depending on the range of key values accessed by the process.

Figure 14-3 shows a clustered index partition scan for a scan that spans three partitions. Worker processes A, B, and C are assigned to each of the table's three partitions. The scan involves two methods:

- Method 1

Worker process A traverses the clustered index to find the first starting page that satisfies the search argument, about midway through Partition 1. It then begins scanning data pages until it reaches the end of Partition 1.

- Method 2

Worker processes B and C do not use the clustered index, but, instead, they begin scanning data pages from the beginning of their partitions. Worker process B completes scanning when it

reaches the end of Partition 2. Worker process C completes scanning about midway through Partition 3, when the data rows no longer satisfy the SARG.

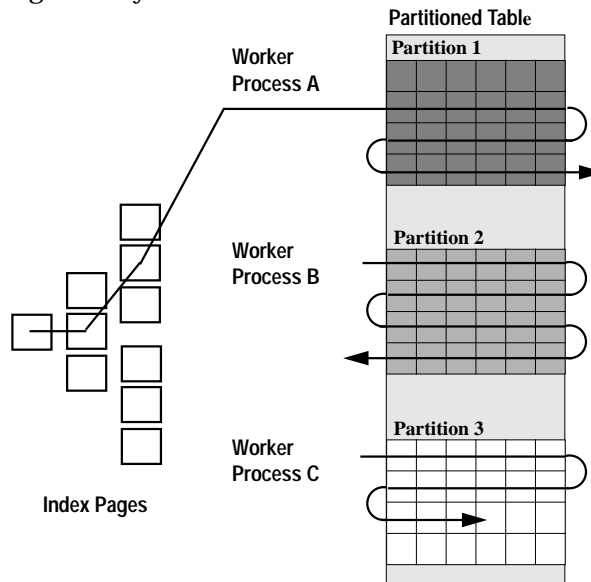


Figure 14-3: Parallel clustered index partition scan

Requirements for Consideration

The optimizer considers a clustered index partition scan only when the following requirements are met:

- The query accesses at least 20 data pages of the table.
- The table is partitioned.
- The table's data is not skewed in relation to the number of partitions. Table data is considered skewed when the size of the largest partition is two or more times the average partition size.

Cost Model

The Adaptive Server optimizer computes the cost of a clustered index partition scan differently, depending on the total number of pages that need to be scanned:

- If the total number of pages that need to be scanned is less than or equal to two times the average size of a partition, then the

optimizer costs the scan as the total number of pages to be scanned divided by 2.

- If the total number of pages that need to be scanned is greater than two times the average size of a partition, then the optimizer costs the scan as the average number of pages in a partition.

The actual cost of the scan may be higher if:

- The total number of pages that need to be scanned is less than the size of a partition, and
- The data to be scanned lies entirely within one partition

If both of these conditions are true, the actual cost of the scan is the same as if the scan were executed serially.

Parallel Nonclustered Index Hash-Based Scan

A nonclustered index hash-based scan is performed in one of two ways, depending on whether the table is a heap table or whether the table has a clustered index. The major difference between the two methods is the hashing mechanism:

- For a table with a clustered index, the hash is on the key values.
- For a table with no clustered index, the scan hashes on the page ID.

In the following list of steps, these two scans differ in step 2, where the hash function is applied to either the page ID or the key:

1. All assigned worker processes traverse the index to find the first leaf-level index page that references data pages with rows that match the SARG.
2. All worker processes scan the leaf-level index pages.
 - If the table is a heap, the worker processes apply a hash function to the data page IDs referenced in each index page. The hash function determines which worker processes scan the rows in each data page. The hash function ensures that only one worker process scans the rows in any given data page in the table.
 - If a clustered index exists on the table, the worker processes apply a hash function to the nonclustered index key value. The hash function determines which data rows will be read by which worker process. The hash function ensures that only one worker process reads any given row.

3. Each worker process reads the rows in its assigned data pages.
4. Steps 2 and 3 are repeated until the referenced data pages in the leaf-level index pages no longer contain rows that match the SARG.

Figure 14-4 illustrates a nonclustered index hash-based scan on a heap table with two worker processes.

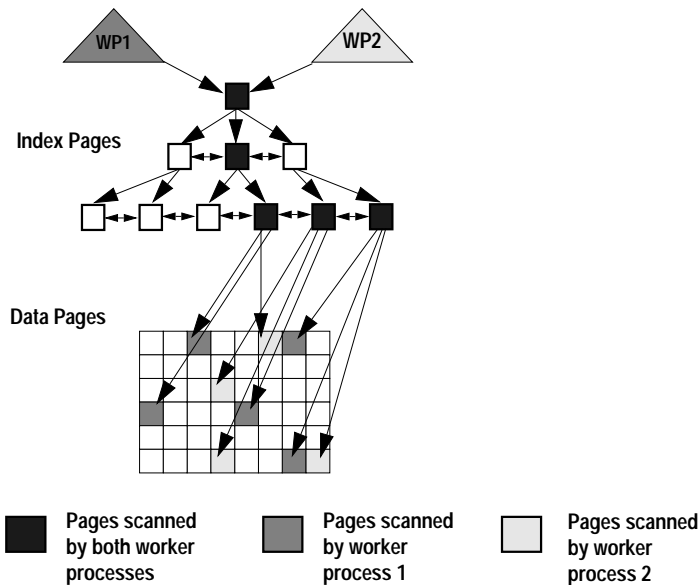


Figure 14-4: Nonclustered index hash-based scan

Cost Model

The cost model of a nonclustered index scan uses the formula:

```
cost of nonclustered index hash-based scan =
# of index pages to scan above leaf level +
# of leaf-level index pages to scan +
(# of data pages referenced in leaf level index
pages / # of worker processes)
```

The optimizer considers a nonclustered index hash-based scan for any tables in a query that have useful nonclustered indexes. The query must also access at least 20 data pages of the table.

For example, if 3 worker processes must scan 3 index pages to find the first leaf-level index page and then scan 5 leaf-level index pages, which point to 60 data pages, the basic cost is $3 + 5 + (60 / 3) = 28$ page I/Os.

► Note

If a nonclustered index covers the result of a query, then the optimizer does not consider using the nonclustered index hash-based scan. See “Index Covering” on page 4-21 for more information about index covering.

Additional Parallel Strategies

In addition to the above access methods, Adaptive Server may employ additional strategies when executing queries in parallel. Those strategies involve the use of partitioned worktables and parallel sorting.

Partitioned Worktables

For queries that require a worktable, Adaptive Server may choose to create a partitioned worktable and populate it using multiple worker processes. Partitioning the worktable improves performance when Adaptive Server populates the table, and, therefore, improves the response time of the query as a whole. See “Parallel Query Examples” on page 14-21 for examples of queries that can benefit from the use of partitioned worktables.

Parallel Sorting

Parallel sorting is a technique that employs multiple worker processes to sort data in parallel, similar to the way multiple worker processes execute a query in parallel. `create index` and any query that requires sorting can benefit from the use of parallel sorting.

The Adaptive Server query optimizer does not directly optimize or control the execution of a parallel sort. See “Parallel Query Examples” on page 14-21 for examples of queries that can benefit from the parallel sorting strategy. Also, see “Overview of the Parallel Sorting Strategy” on page 15-3 for a detailed explanation of how Adaptive Server executes a sort in parallel.

Summary of Parallel Access Methods

The following table summarizes the potential use of parallel access methods in Adaptive Server query processing. In all cases, the query

must access at least 20 data pages in the table before the optimizer considers parallel access methods.

Table 14-1: Parallel access method summary

Parallel Method	Computed Cost (in Page I/Os)	Requirements for Consideration	Competing Serial Methods
Partition scan	Number of pages in the largest partition	Partitioned table with balanced data	Serial table scan, serial index scan
Hash-based table scan	Number of pages in table	Any outer table in a join query and that is a heap	Serial table scan, serial index scan
Clustered index partition scan	If total number of pages to be scanned $\leq 2 * \text{number of pages in average-sized partition}$, then: Total number of pages to be scanned / 2 If total number of pages to be scanned $> 2 * \text{number of pages in average-sized partition}$, then: Average number of pages in a partition	Partitioned table with a useful clustered index	Serial index scan
Nonclustered index hash-based scan	Number of index pages above leaf level to scan + number of leaf-level index pages to scan + (number of data pages referenced in leaf-level index pages / number of worker processes)	Any table with a useful nonclustered index	Serial index scan

Selecting Parallel Access Methods

For a given table in a query, the Adaptive Server optimizer first evaluates the available indexes and partitions to determine which access methods it can use to scan the table's data. Table 14-2 shows the access methods that the optimizer may evaluate for different table and index combinations. Hash-based table scans are considered

only for the outer table in a query, unless the query uses the **parallel** optimizer hint.

Table 14-2: Determining applicable access methods

	No Useful Index	Useful Clustered Index	Useful Nonclustered Index
Partitioned Table	Partition scan Hash-based table scan (if table is a heap) Serial table scan	Clustered index partition scan Serial index scan	Nonclustered index hash-based scan Serial index scan
Unpartitioned Table	Hash-based table scan (if table is a heap) Serial table scan	Serial index scan	Nonclustered index hash-based scan Serial index scan

The optimizer may further eliminate parallel access methods from consideration, based on the number of worker processes that are available to the query. This process of elimination occurs when the optimizer computes the degree of parallelism for the query as a whole. For an example, see “Partitioned Heap Table Example” on page 14-15.

Degree of Parallelism for Parallel Queries

The **degree of parallelism** for a query is the number of worker processes chosen by the optimizer to execute the query in parallel. The degree of parallelism depends both on the upper limit to the degree of parallelism for the query and on the level of parallelism suggested by the optimizer.

The process of computing the degree of parallelism for a query is important for two reasons:

- The final degree of parallelism directly affects the performance of a query since it specifies how many worker processes should do the work in parallel.
- While computing the degree of parallelism, the optimizer disqualifies parallel access methods that would require more worker processes than the limits set by configuration parameters, the **set** command, or the **parallel** clause in a query. This reduces the total number of access methods that the optimizer must consider when costing the query, and, therefore, decreases the overall

optimization time. Disqualifying access methods in this manner is especially important for multitable joins, where the optimizer must consider many different combinations of join orders and access methods before selecting a final query plan.

Upper Limit to Degree of Parallelism

A System Administrator configures the upper limit to the degree of parallelism using server-wide configuration parameters. Session-wide and query-level options can further limit the degree of parallelism. These limits set both the total number of worker processes that can be used in a parallel query and the total number of worker processes that can be used for hash-based access methods.

The optimizer removes from consideration any parallel access methods that would require more worker processes than the upper limit for the query. (If the upper limit to the degree of parallelism is 1, the optimizer does not consider any parallel access methods.) See “Configuration Parameters for Controlling Parallelism” on page 13-12 for more information about configuration parameters that control the upper limit to the degree of parallelism.

Optimized Degree of Parallelism

The optimizer can potentially use worker processes up to the maximum degree of parallelism set at the server, session or query level. However, the optimized degree of parallelism is frequently less than this maximum. The optimizer chooses the degree of parallelism based on the number of partitions in the tables of the query and the number of worker processes configured.

For partition-based access methods, Adaptive Server requires one worker process for every partition in a table. If the number of partitions exceeds `max parallel degree` or a session-level or query-level limit, the optimizer uses a hash-based or serial access method.

For hash-based access methods, the optimizer does not compute an optimal degree of parallelism; instead, it uses the number of worker processes specified by the `max scan parallel degree` parameter. It is up to the System Administrator to set `max scan parallel degree` to an optimal value for the Adaptive Server system as a whole. A general rule of thumb is to set this parameter to no more than 2 or 3, since it takes only 2–3 worker processes to fully utilize the I/O of a given physical device.

Degree of Parallelism Examples

The examples in this section show how the limits to the degree of parallelism affect the following types of queries:

- A partition heap table
- A nonpartitioned heap table
- A table with a clustered index

Partitioned Heap Table Example

Assume that **max parallel degree** is set to 10 worker processes and **max scan parallel degree** is set to 3 worker processes. For a query on a heap table with 6 partitions and no useful nonclustered index, the optimizer costs the following access methods with the corresponding degrees of parallelism:

- A parallel partition scan using 6 worker processes
- A serial index scan using a single process

If **max parallel degree** is set to 5 worker processes, then the optimizer does not consider any parallel access methods for a table with 6 partitions.

The situation changes if the query involves a join: another parallel access method is considered. Assuming again that **max parallel degree** is set to 10 worker processes, the query involves a join, and the table is the outer table in the query, then the optimizer considers the following access methods:

- A partition scan using 6 worker processes
- A serial scan using a single process
- A hash-based table scan using 3 worker processes

If **max scan parallel degree** is set to 5 and **max parallel degree** is set to 3, then the optimizer considers the following access methods:

- A hash-based table scan using 3 worker processes
- A serial scan using a single process

Finally, if **max parallel degree** is set to 5 and **max scan parallel degree** is set to 1, then the optimizer does not consider any parallel access methods.

Nonpartitioned Heap Table Example

If the query involves a join, and **max scan parallel degree** is set to 3, and the nonpartitioned heap table is the outer table in the query, then the optimizer considers the following access methods:

- A hash-based table scan using 3 worker processes
- A serial scan using a single process

If **max scan parallel degree** is set to 1, then the optimizer does not consider any parallel access methods.

See “Single-Table Scans” on page 14-21 for more examples of determining the degree of parallelism for queries.

Tables with Clustered Indexes

The Adaptive Server optimizer considers parallel access methods on tables with clustered indexes only if the table is partitioned. For the optimizer to consider a parallel access method for a table with 6 partitions, **max parallel degree** must be set to at least 6.

The Adaptive Server optimizer considers parallel access methods on tables with clustered indexes only when:

- The table is partitioned
For a table with 6 partitions, **max parallel degree** must be set to at least 6.
- The table has a useful nonclustered index
The optimizer will consider a parallel nonclustered hash-based index scan on a table with a clustered index. **max scan parallel degree** must be set higher than 1.

Run-Time Adjustments to Worker Processes

Even after the optimizer determines a degree of parallelism for the query as a whole, Adaptive Server may make final adjustments at run time to compensate for the actual number of worker processes that are available. If fewer worker processes are available at run time than are suggested by the optimizer, the degree of parallelism is reduced to a level that is consistent with the available worker processes and the access methods in the final query plan. “Run-Time Adjustment of Worker Processes” on page 14-30 describes the process of adjusting the degree of parallelism at run time and explains how to determine when these adjustments occur.

Parallel Strategies for Joins

The Adaptive Server optimizer considers the same parallel access methods when optimizing joins as it does for single-table queries. However, since multiple tables are involved in a join, the optimizer must also consider:

- The join order for each combination of tables in the query and the cost of both serial and parallel methods for table.
- The degree of parallelism for the query as a whole, since more than one parallel access method may be required to scan the data in joined tables.

Join Order and Cost

The optimizer considers the best join order for tables in a query by evaluating all possible combinations of inner and outer tables with all applicable access methods. The join order for a parallel query plan may use a combination of parallel serial access methods for the joined tables. The final join order for a parallel query plan may be different from the join order of a serial query plan.

Determining the Cost of Serial Join Orders

In order to select the best combination of join orders and access methods, the optimizer uses a function to calculate the cost of a given join order. For a serial join of two tables, A and B, the cost of having A as the outer table and B as the inner table is calculated as follows:

```
# of accessed pages of A +
(# of qualifying rows in A *
 # of pages of B to scan for each qualifying row)
```

Determining the Cost of Parallel Join Orders

For tables that are accessed in parallel, the values for “# of pages” and “# of qualifying rows” are the largest number of pages and rows accessed by any one worker process executing the join. So, if table A is accessed using 4 worker processes and table B is accessed serially, the cost of having A as the outer table and B as the inner table is calculated as follows:

```
(# of accessed pages of A / 4) +
(# of qualifying rows in A / 4) *
 # of pages of B for each qualifying row
```

Conversely, the cost of having B as the outer table and A as the inner table, with B accessed serially and A accessed with 4 worker processes, is calculated as follows:

```
# of pages of B +  
# of qualifying rows in B *  
(# of pages of A to scan for each qualifying row/4)
```

Degree of Parallelism for Joins

For individual tables in a multitable join, the optimizer computes the degree of parallelism using the same rules as described under “Optimized Degree of Parallelism” on page 14-14. However, the degree of parallelism for the join query as a whole is the **product** of the worker processes that access individual tables in the join. All worker processes allocated for a join query access all tables in the join. Using the product of worker processes to drive the degree of parallelism for a join ensures that processing is distributed evenly over partitions and that the join returns no duplicate rows.

Figure 14-5 illustrates this rule for two tables in a join where the outer table has 3 partitions and the inner table has 2 partitions. If the optimizer determines that partition-based access methods are to be used on each table, then the query requires a total of 6 worker processes to execute the join. Each of the 6 worker processes scans

one partition of the outer table and one partition of the inner table to process the join condition.

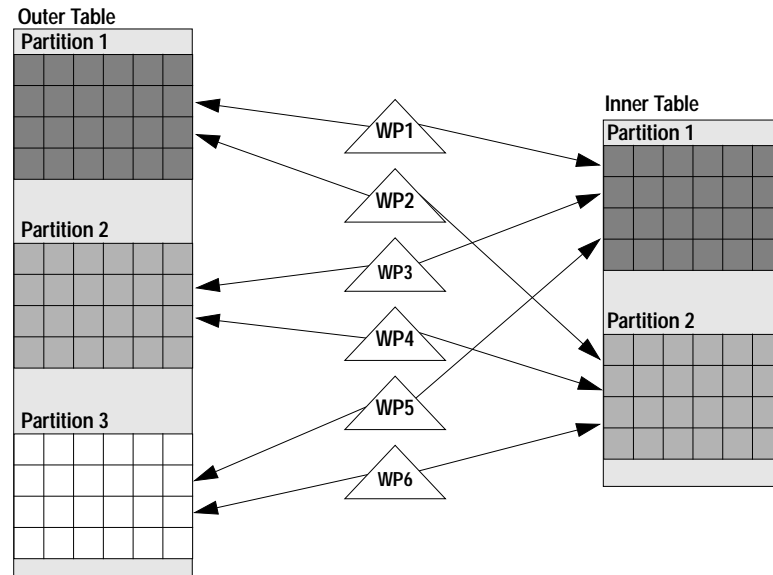


Figure 14-5: Worker process usage for joined tables

In Figure 14-5, if the optimizer chose to scan the inner table using a serial access method, only 3 worker processes would be required to execute the join. In this situation, each worker process would scan one partition of the outer table, and all worker processes would scan the inner table to find matching rows.

Therefore, for any 2 tables in a query with scan degrees of m and n respectively, the potential degrees of parallelism for a join between the two tables are:

- 1, if the optimizer accesses both tables serially
- $m*1$, if the optimizer accesses the first table using a parallel access method (with m worker processes), and the second table serially.
- $n*1$, if the optimizer accesses the second table using a parallel access method (with n worker processes) and the first table serially.
- $m*n$, if the optimizer access both tables using parallel access methods

Alternative Plans

Using partition-based scans on both tables in a join is fairly rare because of the high cost of repeatedly scanning the inner table. The optimizer may also choose:

- The reformatting strategy, if reformatting is a cheaper alternative.
- A partitioned-based scan plus a nonclustered index hash-based scan, when a join returns more 20 or more rows that match the nonclustered index key. See Figure 13-7 on page 13-10 for an illustration.

Computing the Degree of Parallelism for Joins

To determine the degree of parallelism for a join between any two tables (and to disqualify parallel access methods that would require too many worker processes), the optimizer applies the following rules:

1. The optimizer determines possible access methods and degrees of parallelism for the outer table of the join. This process is the same as for single-table queries. See “Optimized Degree of Parallelism” on page 14-14.
2. For each access method determined in step 1, the optimizer calculates the remaining number of worker processes that are available for the inner table of the join. The following formula determines this number:

$$\text{remaining worker processes} = \frac{\text{upper limit of worker processes}}{\text{worker processes required by outer table}}$$

3. The optimizer uses the remaining number of worker processes as an upper limit to determine possible access methods and degrees of parallelism for the inner table of the join.

The optimizer repeats this process for all possible join orders and access methods and applies the cost function for joins to each combination. The optimizer selects the least costly combination of join orders and access methods, and the final combination drives the degree of parallelism for the join query as a whole. See “Multitable Joins” on page 14-23 for examples of this process.

Outer Joins and the Existence Clause

Adaptive Server imposes an additional restriction for join queries that use the exists clause. For these queries, only the number of partitions in the outer table drives the degree of parallelism for the query as a whole; there will be only as many worker processes as there are partitions in the outer table. The inner table in such a query is always accessed serially. “Multitable Joins” on page 14-23 shows an example of how Adaptive Server optimizes joins with the exists clause.

Parallel Query Examples

The following sections further explain and provide examples of how Adaptive Server optimizes these types of parallel queries:

- Single-Table Scans
- Multitable Joins
- Subqueries
- Queries That Require Worktables
- union Queries
- Queries with Aggregates
- select into Statements

Commands that insert, delete, or update data, and commands executed from within cursors, are never considered for parallel query optimization.

Single-Table Scans

The simplest instances of parallel query optimization involve queries that access a single base table. Adaptive Server optimizes these queries by evaluating the base table to determine applicable access methods and then applying cost functions to select the least costly plan.

Understanding how Adaptive Server optimizes single-table queries is integral to understanding more complex parallel queries. Although queries such as multitable joins and subqueries use additional optimization strategies, the process of accessing individual tables for those queries is the same.

The following example shows instances in which the optimizer uses parallel access methods on single-table queries.

Table Partition Scan Example

This example shows a query where the optimizer chooses a table partition scan over a serial table scan. The server configuration and table layout are as follows:

Configuration Parameter Values

Parameter	Setting
max parallel degree	10 worker processes
max scan parallel degree	2 worker processes

Table Layout

Table Name	Useful Indexes	Number of Partitions	Number of Pages
<i>authors</i>	None	5	Partition 1: 50 pages Partition 2: 70 pages Partition 3: 90 pages Partition 4: 80 pages Partition 5: 10 pages

The example query is:

```
select *
  from authors
 where au_lname < "L"
```

Using the logic in Table 14-2 on page 14-13, the optimizer determines that the following access methods are available for consideration:

- Partition scan
- Serial table scan

The optimizer does not consider a hash-based table scan for the table, since the balance of pages in the partitions is not skewed, and the upper limit to the degree of parallelism for the table, 10, is high enough to allow a partition-based scan.

The optimizer computes the cost of each access method, as follows:

```
cost of table partition scan =
# of pages in the largest partition = 90

cost of serial table scan =
# of pages in table = 300
```

The optimizer chooses to perform a table partition scan at a cost of 90 page I/Os. Because the table has 5 partitions, the optimizer chooses to use 5 worker processes. The final `showplan` output for this query optimization is:

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 5 worker
processes.
  STEP 1
    The type of query is SELECT.
    Executed in parallel by coordinating process and 5
    worker processes.
  FROM TABLE
    authors
  Nested iteration.
  Table Scan.
  Ascending scan.
  Positioning at start of table.
  Executed in parallel with a 5-way partition scan.
  Using I/O Size 16 Kbytes.
  With LRU Buffer Replacement Strategy.
  Parallel network buffer merge.
```

Multitable Joins

When optimizing joins, the optimizer considers the best join order for all combinations of tables and applicable access methods. The optimizer uses a different strategy to select access methods for inner and outer tables and the degree of parallelism for the join query as a whole.

As in serial processing, the optimizer weighs many alternatives for accessing a particular table. The optimizer balances the costs of parallel execution with other factors that affect join queries, such as the presence of a clustered index, the possibility of reformatting the inner table, the join order, and the I/O and caching strategy. The following discussion focuses only on parallel vs. serial access method choices.

Parallel Join Optimization and Join Orders

This example illustrates how the optimizer devises a query plan for a join query that is eligible for parallel execution. The server configuration and table layout are as follows:

Configuration Parameter Values

Parameter	Setting
max parallel degree	15 worker processes
max scan parallel degree	3 worker processes

Table Layout

Table Name	Number of Partitions	Number of Pages	Number of Rows
<i>publishers</i>	1 (not partitioned)	1,000	80,000
<i>titles</i>	10	10,000 (distributed evenly over all partitions)	800,000

The example query involves a simple join between these two tables:

```
select *
  from publishers, titles
 where publishers.pub_id = titles.pub_id
```

In theory, the optimizer would consider the costs of all the possible combinations:

- *titles* as the outer table and *publishers* as the inner table, with *titles* accessed in parallel
- *titles* as the outer table and *publishers* as the inner table, with *titles* accessed serially
- *publishers* as the outer table and *titles* as the inner table, with *titles* accessed in parallel
- *publishers* as the outer table and *titles* as the inner table, with *titles* accessed serially
- *publishers* as the outer table and *titles* as the inner table, with *publishers* accessed in parallel

For example, the cost of a join order in which *titles* is the outer table and is accessed in parallel is calculated as follows:

$$\begin{aligned} & (10,000 \text{ pages} / 10 \text{ worker process}) + \\ & (800,000 \text{ rows} / 10 \text{ worker processes}) * \\ & 1,000 \text{ pages per row} = 80,001,000 \text{ pages} \end{aligned}$$

The cost of having *publishers* as the outer table is calculated as follows:

$$\begin{aligned} & 1,000 \text{ pages} + 80,000 \text{ rows} * \\ & (10,000 \text{ pages per row} / 10 \text{ worker processes}) = \\ & 80,001,000 \text{ pages} \end{aligned}$$

And so on. However, other factors are often more important in determining the join order than whether a particular table is eligible for parallel access.

Scenario A: Clustered Index on publishers

The presence of a useful clustered index is often the most important factor in how the query optimizer creates a query plan for a join query. If *publishers* has a clustered index on *pub_id* and *titles* has no useful index, the optimizer chooses the indexed table (*publishers*) as the inner table. With this join order, each access to the inner table takes only a few reads to find rows.

With *publishers* as the inner table, the optimizer costs the eligible access methods for each table. For *titles*, the outer table, it considers:

- A parallel partition scan (cost is number of pages in the largest partition)
- A serial table scan (cost is number of pages in the table).

For *publishers*, the inner table, the optimizer considers only:

- A serial clustered index scan

The final cost of the query is the cost of accessing *titles* in parallel times the number of accesses of the clustered index on *publishers*.

Scenario B: Clustered Index on titles

If *titles* has a clustered index on *pub_id*, and *publishers* has no useful index, the optimizer chooses *titles* as the inner table in the query.

With the join order determined, the optimizer costs the eligible access methods for each table. For *publishers*, the outer table, it considers:

- A hash-based table scan (the initial cost is the same as a serial table scan)

For *titles*, the inner table, the optimizer considers only:

- A serial clustered index scan

In this scenario, the optimizer chooses parallel over serial execution of *publishers*. Even though a hash-based table scan has the same cost as a serial scan, the processing time is cut by one-third, because each worker process can scan the inner table's clustered index simultaneously.

Scenario C: Neither Table Has a Useful Index

If neither table has a useful index, table size and available cache space can be more important factors than potential parallel access for join order. The benefits of having a smaller table as the inner table outweigh the benefits of one parallel access method over the other. The optimizer chooses the *publishers* table as the inner table, because it is small enough to be read once and kept in cache, reducing costly physical I/O.

Then, the optimizer costs the eligible access methods for each table. For *titles*, the outer table, it considers:

- A parallel partition scan (cost is number of pages in the largest partition)
- A serial table scan (cost is number of pages in the table)

For *publishers*, the inner table, it considers only:

- A serial table scan loaded into cache

The optimizer chooses to access *titles* in parallel, because it reduces the cost of the query by a factor of 10.

Note that in some cases where neither table has a useful index, the optimizer chooses the reformatting strategy, creating a temporary table and clustered index instead of repeatedly scanning the inner table.

Subqueries

The Adaptive Server optimizer considers parallel access methods only for the outermost query block in a query containing a subquery; the inner, nested queries always execute serially. If a subquery is flattened into a join, some tables are moved into the outermost query block and are considered for parallel access methods. The optimizer does not consider parallel access methods for materialized subqueries.

Although the optimizer considers parallel access methods only for the outermost query block in a subquery, all worker processes that access the outer query block also access the inner tables of the nested subqueries.

Example of Subqueries in Parallel Queries

This example illustrates a subquery optimized for parallel execution. Assume the *titles* table contains 5 partitions and `max parallel degree` is set to 10. The sample query is:

```
select title_id
   from titles
  where total_sales > all (select total_sales
                        from titles
                        where type = 'business')
```

Adaptive Server optimizes the outermost query block, as it does for any other single-table query (see “Single-Table Scans” on page 14-21). In this example, the outer query scans the *titles* table with a parallel partition scan and 5 worker processes. This access method and degree of parallelism are shown in the first portion of the `showplan` output, and in the subquery block:

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 5 worker
processes.
  STEP 1
    The type of query is SELECT.
    Executed in parallel by coordinating process and 5
worker processes.
    FROM TABLE
      titles
    Nested iteration.
    Table Scan.
    Ascending scan.
    Positioning at start of table.
    Executed in parallel with a 5-way partition scan.
```

```

Run subquery 1 (at nesting level 1).
Using I/O Size 16 Kbytes.
With LRU Buffer Replacement Strategy.
Parallel network buffer merge.
NESTING LEVEL 1 SUBQUERIES FOR STATEMENT 1.
QUERY PLAN FOR SUBQUERY 1 (at nesting level 1 and at line 1).
Correlated Subquery.
Subquery under an ALL predicate.
STEP 1
The type of query is SELECT.
Evaluate Ungrouped ANY AGGREGATE.
Executed by 5 worker processes.
FROM TABLE
    titles
EXISTS TABLE : nested iteration.
Table Scan.
Ascending scan.
Positioning at start of table.
Using I/O Size 16 Kbytes.
With LRU Buffer Replacement Strategy.
END OF QUERY PLAN FOR SUBQUERY 1.

```

Each worker in the inner, nested query block in serial. Although the subquery is run once for each row in the outer table, each worker process performs only one-fifth of the executions. `showplan` output for the subquery indicates that the nested query is “Executed by 5 worker processes,” since each worker process used in the outer query block scans the table specified in the inner query block:

Each worker process maintains a separate cache of subquery results, so the subquery may be executed slightly more often than in serial processing.

Queries That Require Worktables

Parallel queries that require worktables create partitioned worktables and populate them in parallel. For queries that require sorts, the Adaptive Server parallel sort manager determines whether to use a serial or parallel sort. See Chapter 15, “Parallel Sorting,” for more information about parallel sorting.

union Queries

Adaptive Server considers parallel access methods for all parts of a `union` query. If a `union` query requires a worktable, then the worktable may also be partitioned and populated in parallel by worker

processes. See “Recognizing and Managing Run-Time Adjustments” on page 14-32 for sample `showplan` output from a `union` query executed in parallel.

If a `union` query is to return no duplicate rows, then a parallel sort may be performed on the internal worktable to remove duplicate rows. See Chapter 15, “Parallel Sorting,” for more information about parallel sorting.

Queries with Aggregates

Adaptive Server considers parallel access methods for queries that return aggregate results in the same way it does for other queries. For queries that use the `group by` clause to return a grouped aggregate result, Adaptive Server also creates multiple worktables with clustered indexes—one worktable for each worker process that executes the query. Each worker process stores partial aggregate results in its designated worktable. As worker processes finish computing their partial results, they merge those results into a common worktable. After all worker processes have merged their partial results, the common worktable contains the final grouped aggregate result set for the query.

select into Statements

The `select into` command creates a new table to store the query’s result set. Adaptive Server optimizes the base query portion of a `select into` command in the same way it does a standard query, considering both parallel and serial access methods, as described in this chapter. A `select into` statement that is executed in parallel:

1. Creates the new table using columns specified in the `select into` statement.
2. Creates n partitions in the new table, where n is the degree of parallelism that the optimizer chose for the query as a whole.
3. Populates the new table with query results, using n worker processes.
4. Unpartitions the new table.

Performing a `select into` statement in parallel requires additional steps compared to the equivalent serial query plan. Therefore, the execution of a parallel `select into` statement takes place using 4 discrete transactions, rather than the 2 transactions of a serial `select into`

statement. See *select* in the *Adaptive Server Reference Manual* for information about how this change affects the database recovery process.

Parallel *select into* Example

The following *select into* statement creates a new table by scanning all rows in the *authors* table. The *authors* table contains 4 partitions, and the optimizer chooses to scan the table with 4 worker processes. This means that the newly created table, *authors_west*, is first created and then altered to contain 4 partitions before it is populated with data. Adaptive Server unpartitions *authors_west* as the final step, and the *select into* statement ultimately yields a new heap table.

The *select into* statement and *showplan* output are as follows:

```

select *
      into authors_west
      from authors

```

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 4 worker processes.

STEP 1
The type of query is CREATE TABLE.
Executed by coordinating process.

STEP 2
The type of query is INSERT.
The update mode is direct.
Executed in parallel by 4 worker processes.
FROM TABLE
 authors
Nested iteration.
Table Scan.
Ascending scan.
Positioning at start of table.
Executed in parallel with a 4-way partition scan.
Using I/O Size 16 Kbytes.
With LRU Buffer Replacement Strategy.
TO TABLE
 authors_west

Run-Time Adjustment of Worker Processes

The output of *showplan* describes the optimized plan for a given query. An optimized query plan specifies the access methods and the degree of parallelism that the optimizer suggests at the time the

query is compiled. At execution time, there may be fewer worker processes available than are required by the optimized query plan. This can occur when:

- There are not enough worker processes available for the optimized query plan
- The server-level or session-level limits for the query were reduced after the query was compiled. This can happen with queries executed from within stored procedures.

In these circumstances, Adaptive Server may create an adjusted query plan to compensate for the available worker processes. An **adjusted query plan** is a plan generated at run time that compensates for the lack of available worker processes. An adjusted query plan may use fewer worker processes than the optimized query plan, and it may use a serial access method instead of a parallel method for one or more of the tables.

The response time of an adjusted query plan may be significantly longer than its optimized counterpart. Adaptive Server provides:

- A set option, `process_limit_action`, allows you to control whether run-time adjustments are allowed.
- Information on run-time adjustments in `sp_sysmon` output.

How Adaptive Server Adjusts a Query Plan

Adaptive Server uses two basic rules to reduce the number of required worker processes in an adjusted query plan:

1. If the optimized query plan specifies a partition-based access method for a table, but not enough processes are available to scan each partition, the adjusted plan uses a serial access method.
2. If the optimized query plan specifies a hash-based access method for a table, but not enough processes are available to cover the optimized degree of parallelism, the adjusted plan reduces the degree of parallelism to a level consistent with the available worker processes.

To illustrate the first case, assume that an optimized query plan recommends scanning a table's 5 partitions using a partition-based table scan. If only 4 worker processes are actually available at the time the query executes, Adaptive Server creates an adjusted query plan that accesses the table in serial, using a single process.

In the second case, if the optimized query plan recommended scanning the table with a hash-based access method and 5 worker processes, the adjusted query plan would still use a hash-based access method, but with, at the most, 4 worker processes.

Evaluating the Effect of Run-Time Adjustments

Although optimized query plans generally outperform adjusted query plans, the difference in performance is not always significant. The ultimate effect on performance depends on the number of worker processes that Adaptive Server uses in the adjusted plan, and whether or not a serial access method is used in place of a parallel method. Obviously, the most negative impact on performance occurs when Adaptive Server uses a serial access method instead of a parallel access method to execute a query.

The performance of multitable join queries can also suffer dramatically from adjusted query plans, since Adaptive Server does not change the join ordering when creating an adjusted query plan. If an adjusted query plan is executed in serial, the query can potentially perform more slowly than an optimized serial join. This may occur because the optimized parallel join order for a query is different from the optimized serial join order.

Recognizing and Managing Run-Time Adjustments

Adaptive Server provides two mechanisms to help you observe run-time adjustments of query plans. These mechanisms include:

- `set process_limit_action` allows you to abort batches or procedures when run-time adjustments take place or print warnings
- `showplan` prints an adjusted query plan when run-time adjustments occur, and `showplan` is effect

Using *set process_limit_action*

The `process_limit_action` option to the `set` command lets you monitor the use of adjusted query plans at a session or stored procedure level. When you set `process_limit_action` to “abort,” Adaptive Server records Error 11015 error and aborts the query, if an adjusted query plan is required. When you set `process_limit_action` to “warning”, Adaptive Server records Error 11014 but still executes the query.

For example, this command aborts the batch when a query is adjusted at runtime:

```
set process_limit_action abort
```

By examining the occurrences of Errors 11014 and 11015 in the error log, you can determine the degree to which Adaptive Server uses adjusted query plans instead of optimized query plans. To remove the restriction and allow run-time adjustments, use:

```
set process_limit_action quiet
```

See `set` in the *Adaptive Server Reference Manual* for more information about `process_limit_action`.

Using *showplan*

When you use `showplan`, Adaptive Server displays the optimized plan for a given query before it runs the query. When the query plan involves parallel processing, and a run-time adjustment is made, `showplan` also displays the adjusted query plan.

► Note

Adaptive Server does not attempt to execute a query when the `set noexec` is in effect, so run-time plans are never displayed while using this option.

The following example shows how the run-time adjustment of worker processes affects the following query:

```
select * from authors1 union
       select *
       from authors2
       where au_lname > "M"
```

In this example, the *authors1* table has 9 partitions and the *authors2* table is unpartitioned. At compilation time, the Adaptive Server configuration is as follows:

Configuration Parameter Values

Parameter	Setting
max parallel degree	15 worker processes
max scan parallel degree	3 worker processes

The optimized plan for the query uses a parallel partition scan to access *authors1* (using 9 worker processes) and a hash-based partition scan to access *authors2* (using 3 of the 9 worker processes). The total degree of parallelism for the query is 9, as shown in the `showplan` output of the optimized query plan.

At the time this query executes, only 2 worker processes are available. Following the rules described in “How Adaptive Server Adjusts a Query Plan” on page 14-31, Adaptive Server accesses *authors1* serially, since there are not enough processes to perform the partition scan. The *authors2* table is still scanned using a hash-based method, but only 2 worker processes are used. These changes to the query plan are visible in the adjusted query plan, which is displayed immediately after the optimized query plan in the `showplan` output:

```

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 9 worker
processes.
  STEP 1
    The type of query is INSERT.
    The update mode is direct.
    Executed in parallel by coordinating process and 9
    worker processes.
    FROM TABLE
      authors1
    Nested iteration.
    Table Scan.
    Ascending scan.
    Positioning at start of table.
    Executed in parallel with a 9-way partition scan.
    Using I/O Size 2 Kbytes.
    With LRU Buffer Replacement Strategy.
    TO TABLE
      Worktable1.
  STEP 1
    The type of query is INSERT.
    The update mode is direct.
    Executed in parallel by coordinating process and 3
    worker processes.
    FROM TABLE
      authors2
    Nested iteration.
    Table Scan.
    Ascending scan.
    Positioning at start of table.
    Executed in parallel with a 3-way hash scan.
    Using I/O Size 2 Kbytes.
    With LRU Buffer Replacement Strategy.

```

```

      TO TABLE
        Worktable1.
STEP 1
  The type of query is SELECT.
  Executed by coordinating process.
  This step involves sorting.
  FROM TABLE
    Worktable1.
  Using GETSORTED
  Table Scan.
  Ascending scan.
  Positioning at start of table.
  Using I/O Size 2 Kbytes.
  With MRU Buffer Replacement Strategy.

```

AN ADJUSTED QUERY PLAN WILL BE USED FOR STATEMENT 1 BECAUSE NOT ENOUGH WORKER PROCESSES ARE AVAILABLE AT THIS TIME.

ADJUSTED QUERY PLAN:
 QUERY PLAN FOR STATEMENT 1 (at line 1).
 Executed in parallel by coordinating process and 2 worker processes.

```

STEP 1
  The type of query is INSERT.
  The update mode is direct.
  Executed by coordinating process.
  FROM TABLE
    authors1
  Nested iteration.
  Table Scan.
  Ascending scan.
  Positioning at start of table.
  Using I/O Size 2 Kbytes.
  With LRU Buffer Replacement Strategy.
  TO TABLE
    Worktable1.
STEP 1
  The type of query is INSERT.
  The update mode is direct.
  Executed in parallel by coordinating process and 2
  worker processes.
  FROM TABLE
    authors2
  Nested iteration.
  Table Scan.
  Ascending scan.
  Positioning at start of table.
  Executed in parallel with a 2-way hash scan.

```

```
Using I/O Size 2 Kbytes.  
With LRU Buffer Replacement Strategy.  
TO TABLE  
    Worktable1.  
STEP 1  
The type of query is SELECT.  
Executed by coordinating process.  
This step involves sorting.  
FROM TABLE  
    Worktable1.  
Using GETSORTED  
Table Scan.  
Ascending scan.  
Positioning at start of table.  
Using I/O Size 2 Kbytes.  
With MRU Buffer Replacement Strategy.  
The sort for Worktable1 is done in Serial
```

Reducing the Likelihood of Run-Time Adjustments

To reduce the number of run-time adjustments you must increase the number of worker processes that are available to parallel queries. You can do this either by adding more total worker processes to the system or by restricting or eliminating parallel execution for noncritical queries, as follows:

- Use set `parallel_degree` and/or set `scan_parallel_degree` to set session-level limits on the degree of parallelism, or
- Use the query-level `parallel 1` and `parallel N` clauses to limit the worker process usage of individual statements.

To reduce the number of run-time adjustments for system procedures, make sure you recompile the procedures after changing the degree of parallelism at the server or session level. See `sp_recompile` in the *Adaptive Server Reference Manual* for more information.

Checking Run-Time Adjustments with `sp_sysmon`

The system procedure `sp_sysmon` shows how many times a request for worker processes was denied due to a lack of worker processes and how many times the number of worker processes recommended for a query was adjusted to a smaller number. The following sections of the report provide information:

- “Worker Process Management” on page 24-17 describes the output for the number of worker process requests that were requested and denied and the success and failure of memory requests for worker processes.
- “Parallel Query Management” on page 24-20 describes the `sp_sysmon` output that reports on the number of run-time adjustments and locks for parallel queries.

If insufficient worker processes in the pool seems to be the problem, compare the number of worker processes used to the number of worker processes configured. If the maximum number of worker processes used is equal to the configured value for `number of worker processes`, and the percentage of worker process requests denied is greater than 80 percent, increase the value for `number of worker processes` and rerun `sp_sysmon`. If the maximum number of worker processes used is less than the configured value for `number of worker processes`, and the percentage of worker thread requests denied is 0 percent, decrease the value for `number of worker processes` to free memory resources.

Diagnosing Parallel Performance Problems

The following sections provide troubleshooting guidelines for parallel queries. They cover two situations:

- The query runs in serial, when you expect it to run in parallel.
- The query runs in parallel, but does not perform as well as you expect.

Query Does Not Run in Parallel (When You Think It Should)

Possible explanations are:

- The configuration parameter `max parallel degree` is set to 1, or the session-level setting `set parallel_degree` is set to 1, preventing all parallel access.
- The configuration parameter `max scan parallel degree` is set to 1, or the session level setting `set scan_parallel_degree` is set to 1, preventing hash-based parallel access.
- There are insufficient worker threads at execution time. Check for run-time adjustments, using the tools discussed in “Run-Time Adjustments to Worker Processes” on page 14-16.

- The scope of the scan is less than 20 data pages. This can be bypassed with the `parallel` clause.
- The plan calls for a table scan and:
 - The table is not a heap,
 - The table is not partitioned,
 - The partitioning is unbalanced, or
 - The table is a heap but is not the outer table of a join.The last two conditions can be bypassed with the `(parallel)` clause.
- The plan calls for a clustered index scan and:
 - The table is not partitioned or
 - The partitioning is unbalanced. This can be bypassed with the `(parallel)` clause.
- The plan calls for a nonclustered index scan, and the chosen index covers the required columns.
- The table is a temporary table or a system table.
- The table is the inner table of an outer join.
- A limit has been set through the Resource Governor, and all parallel plans exceed that limit in terms of total work.
- The query is a type that is not parallelized, such as an `insert`, `update` or `delete` command, a nested (not the outermost) query, or a cursor.

Parallel Performance Is Not As Good As Expected

Possible explanations are:

- There are too many partitions for the underlying physical devices.
- There are too many devices per controller.
- The `(parallel)` clause has been used inappropriately.
- The `max scan parallel degree` is set too high; the recommend range is 2–3.

Calling Technical Support for Diagnosis

If you cannot diagnose the problem using these hints, the following information will be needed by Sybase Technical Support to determine the source of the problem:

- The table and index schema—create table, alter table...partition, and create index statements are most helpful. Provide output from sp_help if the actual create and alter commands are not available.
- The query.
- The output of query run with commands:
 - dbcc traceon (3604,302, 310)
 - set showplan on
 - set noexec on
- The statistics io output for the query.

Resource Limits for Parallel Queries

The tracking of I/O cost limits may be less precise for partitioned tables than for unpartitioned tables, when Adaptive Server is configured for parallel query processing.

When you query a partitioned table, all the labor in processing the query is divided among the partitions. For example, if you query a table with 3 partitions, the query's work is divided among 3 worker processes. If the user has specified an I/O resource limit with an upper bound of 45, the optimizer assigns a limit of 15 to each worker process.

However, since no two threads are guaranteed to perform the exact same amount of work, the parallel processor cannot precisely distribute the work among worker processes. You could get an error message saying you have exceeded your I/O resource limit when, according to showplan or statistics io output, you actually had not. Conversely, one partition may exceed the limit slightly, without the limit taking effect.

See Chapter 12, "Limiting Access to Server Resources," in the *System Administration Guide* for more information about setting resource limits.

15

Parallel Sorting

This chapter discusses how to configure the server for improved performance for commands that perform parallel sorts. The process of sorting data is an integral part of any database management system. Sorting is for creating indexes and for processing complex queries. The Adaptive Server parallel sort manager provides a high-performance, parallel method for sorting data rows. All Transact-SQL commands that require an internal sort can benefit from the use of parallel sorting.

This chapter explains how parallel sorting works and what factors affect the performance of parallel sorts. You need to understand these subjects in order to get the best performance from parallel sorting, and in order to keep parallel sort resource requirements from interfering with other resource needs.

This chapter contains the following sections.

- Commands That Benefit from Parallel Sorting 15-1
- Parallel Sort Requirements and Resources Overview 15-2
- Overview of the Parallel Sorting Strategy 15-3
- Configuring Resources for Parallel Sorting 15-6
- Recovery Considerations 15-17
- Tools for Observing and Tuning Sort Behavior 15-18
- Using `sp_sysmon` to Tune Index Creation 15-23

Commands That Benefit from Parallel Sorting

Any Transact-SQL command that requires data row sorting can benefit from parallel sorting techniques. These commands are:

- `create index` commands and the `alter table...add constraint` commands that build indexes, unique and primary key
- Queries that use the `order by` clause
- Queries that use `distinct`
- Queries that use `union` (except `union all`)
- Queries that use the **reformatting** strategy

In addition, any cursors that use the above commands can benefit from parallel sorting.

Parallel Sort Requirements and Resources Overview

Like parallel query processing, parallel sorting requires more resources than performing the same command in parallel. Response time for creating the index or sorting query results improves, but the server performs more work due to overhead.

Adaptive Server's sort manager determines whether the resources required to perform a sort operation in parallel are available, and also whether a serial or parallel sort should be performed, given the size of the table and other factors. For a parallel sort to be performed, certain criteria must be met:

- The `select into/bulk copy/pllsort` database option must be set to `true` with `sp_dboption` in the target database. For worktable sorts, this option needs to be enabled in `tempdb` (it is enabled by default). For indexes, the option must be enabled in the database where the table resides. For creating a clustered index on a partitioned table, this option must be enabled, or the sort fails. For sorting worktables and creating other indexes, serial sorts can be performed if parallel sorts cannot be performed.
- Parallel sorts must have a minimum number of worker processes available. The number depends on the number of partitions on the table and/or the number of devices on the target segment. The degree of parallelism at the server and session level must be high enough for the sort to use at least the minimum number of worker processes required for a parallel sort. Clustered indexes on partitioned tables must be created in parallel; other sorts can be performed in serial, if enough worker processes are not available. "Worker Process Requirements During Parallel Sorts" on page 15-6 and "Worker Process Requirements for select Query Sorts" on page 15-9.
- For `select` commands that require sorting, and for creating nonclustered indexes, the table to be sorted must be at least eight times the size of the available sort buffers (the value of the `number of sort buffers` configuration parameter), or the sort will be performed in serial mode. This ensures that Adaptive Server does not perform parallel sorting on smaller tables that would not show significant improvements in performance. This rule does not apply to creating clustered indexes on partitioned tables, since this operation always requires a parallel sort. See "Sort Buffer Configuration Guidelines" on page 15-11.
- For `create index` commands, the value of the `number of sort buffers` configuration parameter must be at least as large as the number

of worker processes available for the parallel sort. See “Sort Buffer Configuration Guidelines” on page 15-11.

► **Note**

The **dump transaction** command cannot be used after indexes are created using a parallel sort. You must dump the database. Serial **create index** commands can be recovered, but are recovered by completely re-doing the indexing command, which can greatly lengthen recovery time. Performing database dumps after serial create indexes is recommended to speed recovery, although it is not required in order to use **dump transaction**.

Overview of the Parallel Sorting Strategy

Like the Adaptive Server optimizer, the Adaptive Server parallel sort manager analyzes the available worker processes, the input table, and other resources to determine the number of worker processes to use for the sort.

After determining the number of worker processes to use, Adaptive Server executes the parallel sort. The process of executing a parallel sort is the same for **create index** commands and queries that require sorts. Adaptive Server executes a parallel sort by:

1. Sampling data rows from the input table to determine the distribution of the data. The sample is used to create a distribution map.
2. Reading the table data and dynamically partitioning the key values into a set of sort buffers, as determined by the distribution map.
3. Sorting each individual range of key values and creating subindexes.
4. Merging the sorted subindexes into the final result set.

Each of these steps is described in the sections that follow.

Figure 15-1 depicts a parallel sort of a table with two partitions and two physical devices on its segment.

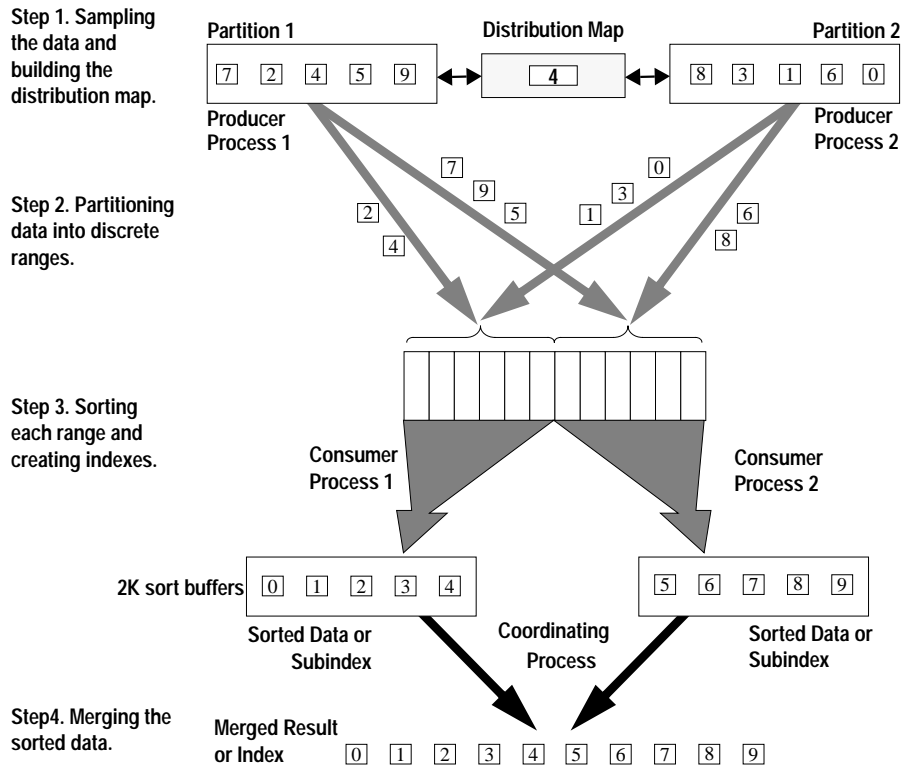


Figure 15-1: Parallel sort strategy

Sampling and Creating a Distribution Map

As a first step in executing a parallel sort, Adaptive Server selects and sorts a random sample of data from the input table. Using the results of this initial sort, Adaptive Server estimates the distribution of values in the input table. This distribution information—referred to as the **distribution map**—is used in the second sort step to divide the input data into equally sized ranges during the next phase of the parallel sort process.

The distribution map contains a key value for the highest key that will be assigned to each range, except the final range in the table. In Figure 15-1, the distribution map shows that all values less than or

equal to 4 are assigned to the first range and that all values greater than 4 are assigned to the second range.

Dynamic Range Partitioning

After creating the distribution map, Adaptive Server employs two kinds of worker processes to perform different parts of the sort. These worker processes are called **producer processes** and **consumer processes**:

- Producer processes read data from the input table and use the distribution map to determine the range to which each key value belongs. The producers distribute the data by copying it to the 2K sort buffers belonging to the correct range.
- Each consumer process reads the data from a range of the sort buffers and sorts it into subindexes, as described in “Range Sorting” on page 15-5.

In Figure 15-1, two producer processes read data from the input table. Each producer process scans one table partition and distributes the data into ranges using the distribution map. For example, the first producer process reads data values 7, 2, 4, 5, and 9. Based on the information in the distribution map, the process distributes values 2 and 4 to the first consumer process and values 7, 5, and 9 to the second consumer process.

Range Sorting

Each partitioned range has a dedicated consumer process that sorts the data in that range independently of other ranges. For queries, the consumer process simply orders the data in the range from the smallest value to the largest. For `create index` commands, the consumer process also builds an index, referred to as a subindex, on the sorted data.

Depending on the size of the table and the number of buffers available to perform the sort, the consumers may perform multiple merge runs, writing intermediate results to disk, and reading and merging those results, until all of the data for the assigned range is completely sorted.

For `create index` commands, each consumer for each partitioned range of data writes to a separate database device. This improves performance through increased I/O parallelism, if database devices reside on separate physical devices and controllers.

Merging Results

After all consumer processes have finished sorting the data for each partitioned range, the **coordinating process** merges the results of the consumer processes into a single result set. For queries, the result set is the final, sorted data that is returned to the client. For a `create index` statement, the coordinating process merges the subindexes into one final index.

Configuring Resources for Parallel Sorting

The following sections describe the different resources used by Adaptive Server when sorting data in parallel:

- Worker processes read the data and perform the sort.
- Sort buffers pass data in cache from producers to consumers, reducing physical I/O.
- Large I/O pools in the cache used for the sort also help reduce physical I/O.
- Multiple physical devices increase I/O parallelism and help to determine the number of worker processes for most sorts.

Worker Process Requirements During Parallel Sorts

To perform parallel sorts, Adaptive Server requires a certain minimum number of worker processes. Many sorts can use additional worker processes to perform the sort more quickly. The minimum number required and the maximum number that can be used are determined:

- By the number of partitions on the table, for clustered indexes
- By the number of devices, for nonclustered indexes
- By the number of devices in *tempdb*, for queries that require sorts

If the minimum number of worker processes is not available:

- Sorts for clustered indexes on partitioned tables must be performed in parallel; they will fail if not enough worker processes are available.
- All other sorts for queries can be performed in serial: sorts for nonclustered indexes, sorts for clustered indexes on unpartitioned tables, and sorts for queries.

The availability of worker processes is limited by server-wide and session-wide limits. At the server level, the configuration parameters `number of worker processes` and `max parallel degree` limit the total size of the pool of worker processes and the maximum number that can be used by any `create index` or `select` command.

The available processes at run time may be smaller than the configured value of `max parallel degree` or the session limit, due to other queries running in parallel. The decision on the number of worker processes to use for a sort is made by the sort manager, not by the optimizer. Since the sort manager makes this decision at run time, and not in an earlier, completely separate step, as in query optimization, parallel sort decisions are based on the actual number of worker processes available when the sort begins.

See “Controlling the Degree of Parallelism” on page 13-10 for more information about controlling the server-wide and session-wide limits.

Worker Process Requirements for Creating Indexes

Table 15-1 shows the number of producers and consumers required to create indexes. The **target segment** for a sort is the segment where the index will be stored when the `create index` command completes. When you create an index, you can specify the location with the `segment_name` clause. If you do not specify a segment, the index is stored on the *default* segment.

Table 15-1: Number of producers and consumers used for create index

Index Type	Producers	Consumers
Nonclustered index	Number of partitions, or 1	Number of devices on target segment
Clustered index on unpartitioned table	1	Number of devices on target segment
Clustered index on partitioned table	Number of partitions, or 1	Number of partitions

Consumers are the workhorses of parallel sort, using CPU time to Perform the actual sort and using I/O to read and write intermediate results and to write the final index to disk. First, the sort manager assigns one worker process as a consumer for each target device. Next, if there are enough available worker processes, the sort manager assigns one producer to each partition in the table. If there

are not enough worker processes to assign one producer to each partition, the entire table is scanned by a single producer.

Clustered Indexes on Partitioned Tables

To create a clustered index on a partitioned table, Adaptive Server requires at least one consumer process for every partition on the table, plus one additional worker process to scan the table. If fewer worker processes are available, then the `create clustered index` command fails and prints a message showing the available and required numbers of worker processes.

If enough worker processes are available, the sort manager assigns one producer process per partition, as well as one consumer process for each partition. This speeds up the reading of the data.

Minimum	1 consumer per partition, plus 1 producer
Maximum	2 worker processes per partition
Can be performed in serial	No

Clustered Indexes on Unpartitioned Tables

Only one producer process can be used to scan the input data for unpartitioned tables. The number of consumer processes is determined by the number of devices on the segment where the index is to be stored. If there are not enough worker processes available, the sort can be performed in serial.

Minimum	1 consumer per device, plus 1 producer
Maximum	1 consumer per device, plus 1 producer
Can be performed in serial	Yes

Nonclustered Indexes

The number of consumer processes is determined by the number of devices on the target segment. If there are enough worker processes available and the table is partitioned, one producer process is used for each partition on the table; otherwise, a single producer process

scans the entire table. If there are not enough worker processes available, the sort can be performed in serial.

Minimum	1 consumer per device, plus 1 producer
Maximum	1 consumer per device, plus 1 producer per partition
Can be performed in serial	Yes

Using *with consumers* While Creating Indexes

RAID devices appear to Adaptive Server as a single database device, so, although the devices may be capable of supporting the I/O load of parallel sorts, Adaptive Server assigns only a single consumer for the device, by default.

The *with consumers* clause to the *create index* statement provides a way to specify the number of consumer processes that *create index* can use. By testing the I/O capacity of striped devices, you can determine the number of simultaneous processes your RAID device can support and use this number to suggest a degree of parallelism for parallel sorting. As a baseline, use one consumer for each underlying physical device. This example specifies eight consumers:

```
create index order_ix on orders (order_id)
with consumers = 8
```

You can also use the *with consumers* clause with the *alter table...add constraint* clauses that create indexes, primary key and unique:

```
alter table orders
add constraint prim_key primary key (order_id)
with consumers = 8
```

The *with consumers* clause can be used for creating indexes—you cannot control the number of consumer processes used in internal sorts for parallel queries. You cannot use this clause when creating a clustered index on a partitioned table. When creating a clustered index on a partitioned table, Adaptive Server must use one consumer process for every partition in the table to ensure that the final, sorted data is distributed evenly over partitions.

Adaptive Server ignores the *with consumers* clause if the specified number of processes is higher than the number of available worker processes or if the specified number of processes exceeds the server or session limits for parallelism.

Worker Process Requirements for *select* Query Sorts

For sorts on worktables, the number of devices in the *system* segment of *tempdb* determines the number of consumers, as follows:

- One consumer process is assigned for each device in *tempdb*, and a single producer process scans the worktable.
- If there is only one device in *tempdb*, the sort is performed using two consumers and one producer.
- If there more devices in *tempdb* than the available worker processes when the sort starts, the sort is performed in serial.

Queries that require worktable sorts have multistep query plans. During the phase of the query where data is selected into the worktable, each worker process selects data into a separate partition of the worktable. At the end of this step, the table is unpartitioned and then scanned by a single producer process.

The determination of the number of worker processes for a worktable sort is made after the scan of the base table completes. Additional worker processes are allocated to perform the sort step. *showplan* does not report this value; the sort manager reports only whether the sort is performed in serial or parallel. The worker processes that scan the table do not participate in the sort, but remain allocated to the parallel task until the sort completes.

Caches, Sort Buffers, and Parallel Sorts

Optimal cache configuration and an optimal setting for the number of sort buffers configuration parameter can greatly speed the performance of parallel sorts. The tuning options to consider when you work with parallel sorting are:

- Cache bindings
- Sort buffers
- Large I/O

In most cases, the configuration you choose for normal run time operation should be aimed at the needs of queries that perform worktable sorts. You need to understand how many simultaneous sorts are needed and the approximate size of the worktables, and then configure the cache used by *tempdb* to optimize the sort.

In many installations, dropping and re-creating indexes is a maintenance activity scheduled during periods of low system usage,

leaving you free to optimize the reindexing by reconfiguring caches and pools and changing cache bindings. If you need to perform index maintenance while users are active, you need to consider the impact that such a reconfiguration could have on user response time. Configuring a large percentage of the cache for exclusive use by the sort or temporarily unbinding objects from caches can seriously impact performance for other tasks.

Cache Bindings

Sorts take place in the cache to which a table is bound. If the table is not bound to a cache, but the database is bound, then that cache is used. If there is no explicit cache binding, the default data cache is used. To configure the number of sort buffers and large I/O for a particular sort, always check the cache bindings. You can see the binding for a table with `sp_help`. To see all of the cache bindings on a server, use `sp_helpcache`.

Once you have determined the cache binding for a table, check the space in the 2K and 16K pools in the cache with `sp_cacheconfig`.

How the Number of Sort Buffers Affects Sort Performance

During the range partitioning step of a sort, producers perform disk I/O to read the input table and consumers perform disk I/O to read and write intermediate sort results to and from disk. The process of passing data from producers to consumers avoids disk I/O by using the sort buffers and copying data rows completely in memory. The reserved buffers are not relinked and washed like buffers during normal processing and are not available to any other tasks for the duration of the sort.

The configuration parameter `number of sort buffers` determines the maximum size of the space that can be used by producers and consumers. These buffers are also used to perform serial sorts. Each sort instance can use up to the `number of sort buffers` value for each sort. If active sorts have reserved all of the buffers in a cache, and another sort needs sort buffers, that sort waits until buffers are available in the cache.

Sort Buffer Configuration Guidelines

Since `number of sort buffers` controls the amount of data that can be read and sorted in one batch, configuring more sort buffers increases the batch size, reduces the number of merge runs needed, and makes the

sort run faster. Changing **number of sort buffers** is dynamic, so you do not have to restart the server.

Some general guidelines for configuring sort buffers are as follows:

- The sort manager chooses serial sorts when the number of pages in a table is less than 8 times the value of **number of sort buffers**. In most cases, the default value (500) works well for select queries and small indexes. At this setting, the sort manager chooses serial sorting for all **create index** and worktable sorts of 4000 pages or less, and parallel sorts for larger result sets, saving worker processes for query processing and larger sorts. It also allows multiple sort processes to use up to 500 sort buffers simultaneously.

A temporary worktable would need to be very large before you would need to set the value higher to reduce the number of merge runs for a sort. See “Estimating Table Sizes in tempdb” on page 19-5 for more information.

- If you are creating indexes on large tables while other users are active, you need to configure the number of sort buffers so that you do not disrupt other activity that needs to use the data cache.
- If you are re-creating indexes during scheduled maintenance periods when few users are active on the system, you may want to configure a high value for sort buffers. To speed your index maintenance, you may want to benchmark performance of high sort buffer values, large I/O, and cache bindings to optimize your index activity.
- The reduction in merge runs is a logarithmic function. Increasing the value of **number of sort buffers** from 500 to 600 has very little effect on the number of merge runs. Increasing the size to a much larger value, such as 5000, can greatly speed the sort by reducing the number of merge runs and the amount of I/O needed.

When more sort buffers are configured, fewer intermediate steps and merge runs need to take place during a sort, and physical I/O is required. When **number of sort buffers** is equal to or greater than the number of pages in the table, the sort can be performed completely in cache, with no physical I/O for the intermediate steps: the only I/O required will be the I/O to read and write the data and index pages.

Using Less Than the Configured Number of Sort Buffers

There are two types of sorts that may use fewer than the configured number of sort buffers:

- Creating a clustered index on a partition table always requires a parallel sort. If the table size is smaller than the number of configured sort buffers, then the sort reserves the number of pages in the table for the sort.
- Small serial sorts reserve just the number of sort buffers required to hold the table in cache.

Configuring the *number of sort buffers* Parameter

When creating indexes in parallel, the number of sort buffers must be equal to or less than 90 percent of the number of buffers in the 2K pool area, before the wash marker, as shown in Figure 15-2

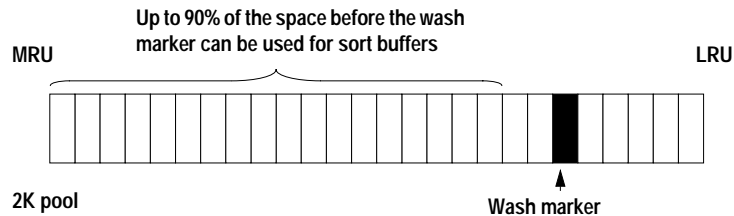


Figure 15-2: Area available for sort buffers

The limit of 90 percent of the pool size is not enforced when you configure the *number of sort buffers* parameter, but it is enforced when you run the *create index* command, since the limit is enforced on the pool for the table being sorted. The maximum value that can be set for *number of sort buffers* is 32,767; this value is enforced by *sp_configure*.

Computing the Allowed Sort Buffer Value for a Pool

sp_cacheconfig returns the size of the pool in megabytes and the wash size in kilobytes. For example, this output shows the size of the pools in the default data cache:

```
Cache: default data cache, Status: Active, Type: Default
      Config Size: 0.00 Mb, Run Size: 15.59 Mb
      Config Replacement: strict LRU, Run Replacement: strict
LRU
```

IO Size	Wash Size	Config Size	Run Size	APF Percent
2 Kb	2578 Kb	4.00 Mb	12.59 Mb	10
16 Kb	512 Kb	3.00 Mb	3.00 Mb	10

This procedure takes the size of the 2K pool and its wash size as parameters, converts both values to pages and computes the maximum number of pages that can be used for sort buffers:

```
create proc bufs @poolsize numeric(6,2), @wash int
as
select "90% of non-wash 2k pool",
      ((@poolsize * 512) - (@wash/2)) * .9
```

The following example executes bufs with values of “12.59 Mb” for the pool size and “2578 Kb” for the wash size:

```
bufs 12.59, 2578
```

The bufs procedure returns the following results:

```
-----
90% of non-wash 2k pool          4641.372000
```

This command sets the number of sort buffers to 4641 pages:

```
sp_configure "number of sort buffers", 4641
```

If the table on which you want to create the index is bound to a user-defined cache, you need to configure the appropriate number of sort buffers for the specific cache. As an alternative, you can unbind the table from the cache, create the index, and rebind the table:

```
sp_unbindcache pubtune, titles
create clustered index title_ix
  on titles (title_id)
sp_bindcache pubtune_cache, pubtune, titles
```

◆ **WARNING!**

The buffers used by a sort are reserved entirely for the use of the sort until the sort completes. They cannot be used by another other task on the server. Setting the number of sort buffers to 90 percent of the pool size can seriously affect query processing if you are creating indexes while other transactions are active.

Procedure for Estimating Merge Runs and I/O

The following procedure estimates the number of merge runs and the amount of physical I/O required to create an index:

```

create proc merge_runs @pages int, @bufs int, @consumers smallint
as
declare @runs smallint, @merges smallint

select @runs = (@pages/@consumers)/(@bufs/@consumers)
if @runs <=1
    select @merges = 0
else
    select @merges = ceiling(log10(@runs) / log10(8.0))

select @merges "Merge Runs",
       2*(@pages/(@bufs/@consumers)) *
       @merges *(@bufs/@consumers) + @pages "Total IO"

```

The parameter for the procedure are:

- *pages*—the number of pages in the table, or the number of leaf-level pages in a nonclustered index.
- *bufs*—the number of sort buffers to configure
- *consumers*—the number of consumers used to perform the sort

This example uses the default number of sort buffers and 20 consumers for a table with 122000 pages:

```
merge_runs 122000, 500, 20
```

The `merge_runs` procedure estimates that 3 merge runs and 854,000 I/Os would be required to create the index:

```

Merge Runs Total IO
-----
          3      854000

```

Increasing the number of sort buffers to 3000 reduces the number of merge runs and the I/O required:

```
merge_runs 122000, 3000, 20
```

```

Merge Runs Total IO
-----
          2      609800

```

The total I/O predicted by this procedure may be different than the I/O usage on your system, depending on the size and configuration of the cache and pools used by the sort.

Configuring Caches for Large I/O During Parallel Sorting

Sorts can use large I/O:

- During the sampling phase

- For the producers scanning the input tables
- For the consumers performing disk I/O on intermediate and final sort results

For these steps, sorts can use the largest pool size available in the cache used by the table being sorted; they can use the 2K pool if no large I/O buffers are available.

Balancing Sort Buffers and Large I/O Configuration

Configuring a pool for 16K buffers in the cache used by the sort greatly speeds I/O for the sort, reducing the number of physical I/Os for a sort by substantially. Part of this I/O savings results from using large I/O to scan the input table.

Additional I/O, both reads and writes, takes place during the merge phases of the sort. The amount of I/O during this step depends on the number of merge phases required. During the sort and merge step, buffers are either read once and not needed again, or they are filled with intermediate sort output results, written to disk, and available for reuse. The cache-hit ratio during sorts will always be low so configuring a large, 16K cache wastes space that can better be used for sort buffers, to reduce merge runs.

For example, creating a clustered index on a 250MB table using a 32MB cache performed optimally with only 4MB configured in the 16K pool and 10,000 sort buffers. Larger pool sizes did not affect the cache hit ratio or number of I/Os. Changing the wash size for the 16K pool to the maximum allowed wash size helped performance slightly, since the small pool size tended to allow buffers to reach the LRU end of the cache before the writes are completed. The following formula computes the maximum allowable wash size for a 16K pool:

```
select floor((size_in_MB * 1024 /16) * .8) * 16
```

Disk Requirements

Disk requirements for parallel sorting are as follows:

- Space is needed to store the completed index.
- Having multiple devices in the target segment increases the number of consumers for worktable sorts and for creating nonclustered indexes and clustered indexes on nonpartitioned tables.

Space Requirements for Creating Indexes

Creating indexes requires space to store the sorted index. For clustered indexes, this requires copying the data rows to new locations in the order of the index key. The newly ordered data rows and the upper levels of the index must be written before the base table can be removed. Unless you are using the `with sorted_data` clause to suppress the sort, creating a clustered index requires approximately 120 percent of the space occupied by the table.

Creating a nonclustered index requires space to store the new index. To help determine the size of objects and the space that is available, use the following system procedures:

- `sp_spaceused`—to see the size of the table. “Using `sp_spaceused` to Display Object Size” on page 6-4.
- `sp_estspace`—to predict the size of the index. See “Using `sp_estspace` to Estimate Object Size” on page 6-9.
- `sp_helpsegment`—to see space left on a database segment. See “Checking Data Distribution on Devices with `sp_helpsegment`” on page 17-27.

Space Requirements for Worktable Sorts

Queries that sort worktables (`order by`, `distinct`, `union`, and reformatting) first copy the needed columns for the query into the worktable and then perform the sort. These worktables are stored on the *system* segment in *tempdb*, so this is the target segment for queries that require sorts. To see the space available and the number of devices, use:

```
tempdb..sp_helpsegment system
```

The process of inserting the rows into the worktable and the parallel sort do not require multiple devices to operate in parallel. However, performance improves when the *system* segment in *tempdb* spans multiple database devices.

Number of Devices in the Target Segment

As described in “Worker Process Requirements During Parallel Sorts” on page 15-6, the number of devices in the target segment determines the number of consumers for all sort operations, except for creating a clustered index on a partitioned table.

Performance considerations for query processing, such as the improvements in I/O when indexes are on separate devices from the data are more important in determining your device allocations and object placement than sort requirements.

If your worktable sorts are large enough to require parallel sorts, multiple devices in the system segment of *tempdb* will speed these sorts, as well as increase I/O parallelism while rows are being inserted into the worktable.

Recovery Considerations

Creating indexes is a minimally logged database operation. Serial sorts are recovered from the transaction log by completely re-doing the sort. However, parallel create index commands are not recoverable from the transaction log—after performing a parallel sort, you must dump the database before you can use the `dump transaction` command on the database.

Adaptive Server does not automatically perform parallel sorting for create index commands unless the `select into/bulk copy/pllsort` database option is set on. Creating a clustered index on a partitioned table always requires a parallel sort; other sort operations can be performed in serial if the `select into/bulk copy/pllsort` option is not enabled.

Tools for Observing and Tuning Sort Behavior

Adaptive Server provides several tools for working with sort behavior:

- `set sort_resources on` shows how a create index command would be performed, without creating the index. See “Using set `sort_resources on`” on page 15-18.
- Several system procedures can help estimate the size, space, and time requirements:
 - `sp_configure`—displays configuration parameters. See “Configuration Parameters for Controlling Parallelism” on page 13-10.
 - `sp_helppartition`—displays information about partitioned tables. See “Getting Information About Partitions” on page 17-26.

- `sp_helpsegment`—displays information about segments, devices, and space usage. See “Checking Data Distribution on Devices with `sp_helpsegment`” on page 17-27.
- `sp_sysmon`—reports on many system resources used for parallel sorts, including CPU utilization, physical I/O, and caching. See “Using `sp_sysmon` to Tune Index Creation” on page 15-23.

Using `set sort_resources on`

The `set sort_resources on` command can help you understand how the sort manager performs parallel sorting for `create index` statements. You can use it before creating an index to determine whether you want to increase configuration parameters or specify additional consumers for a sort.

After you use `set sort_resources on`, Adaptive Server does not actually create indexes, but analyzes resources, performs the sampling step, and prints detailed information about how Adaptive Server would use parallel sorting to execute the `create index` command. Table 15-2 describes the messages that can be printed for sort operations.

Table 15-2: Basic sort resource messages

Message	Explanation	See
The Create Index is done using <code>sort_type</code>	<code>sort_type</code> is either “Parallel Sort” or “Serial Sort.”	“Parallel Sort Requirements and Resources Overview” on page 15-2
Sort buffer size: <i>N</i>	<i>N</i> is the configured value for the number of sort buffers configuration parameter.	“Sort Buffer Configuration Guidelines” on page 15-11
Parallel degree: <i>N</i>	<i>N</i> is the maximum number of worker processes that the parallel sort can use, as set by configuration parameters.	“Caches, Sort Buffers, and Parallel Sorts” on page 15-10
Number of output devices: <i>N</i>	<i>N</i> is the total number of database devices on the target segment.	“Disk Requirements” on page 15-16
Number of producer threads: <i>N</i>	<i>N</i> is the optimal number of producer processes determined by the sort manager.	“Worker Process Requirements During Parallel Sorts” on page 15-6

Table 15-2: Basic sort resource messages (continued)

Message	Explanation	See
Number of consumer threads: N	N is the optimal number of consumer processes determined by the sort manager.	“Worker Process Requirements During Parallel Sorts” on page 15-6
The distribution map contains M element(s) for N partitions.	M is the number of elements that define range boundaries in the distribution map. N is the total number of partitions (ranges) in the distribution map.	“Sampling and Creating a Distribution Map” on page 15-4
Partition Element: N value	N is the number of the distribution map element. <i>value</i> is the distribution map element that defines the boundary of each partition.	“Sampling and Creating a Distribution Map” on page 15-4
Number of sampled records: N	N is the number of sampled records used to create the distribution map.	“Sampling and Creating a Distribution Map” on page 15-4

Sort Examples

The following examples show the output of the `set sort_resources` command.

Nonclustered Index on a Nonpartitioned Table

This example shows how Adaptive Server performs parallel sorting for a `create index` command on an unpartitioned table. Pertinent details for the example are:

- The *default* segment spans 4 database devices
- `max parallel degree` is set to 20 worker processes
- `number of sort buffers` is set to the default, 500 buffers

The following commands set `sort_resources` on and issue a `create index` command on the `orders` table:

```
set sort_resources on
create index order_ix on orders (order_id)
```

Adaptive Server prints the following output:

```

The Create Index is done using Parallel Sort
Sort buffer size: 500
Parallel degree: 20
Number of output devices: 4
Number of producer threads: 1
Number of consumer threads: 4
The distribution map contains 3 element(s) for 4
partitions.
Partition Element: 1

458052

Partition Element: 2

909063

Partition Element: 3

1355747

Number of sampled records: 2418

```

In this example, the 4 devices on the *default* segment determine the number of consumer processes for the sort. Because the input table is not partitioned, the sort manager allocates only 1 producer process, for a total degree of parallelism of 5.

The distribution map uses 3 dividing values for the 4 ranges. The lowest input values up to and including the value 458052 belong to the first range. Values greater than 458052 and less than or equal to 909063 belong to the second range. Values greater than 909063 and less than or equal to 1355747 belong to the third range. Values greater than 1355747 belong to the fourth range.

Nonclustered Index on a Partitioned Table

This example uses the same tables and devices as the first example. However, in this example, the input table is partitioned before creating the nonclustered index. The commands are:

```

set sort_resources on
alter table orders partition 9
create index order_ix on orders (order_id)

```

In this case, the `create index` command under the `sort_resources` option prints the output:

```
The Create Index is done using Parallel Sort
Sort buffer size: 500
Parallel degree: 20
Number of output devices: 4
Number of producer threads: 9
Number of consumer threads: 4
The distribution map contains 3 element(s) for 4
partitions.
Partition Element: 1
```

```
458464
Partition Element: 2
```

```
892035
Partition Element: 3
```

```
1349187
Number of sampled records: 2448
```

Because the input table is now partitioned, the sort manager allocates 9 producer threads, for a total of 13 degrees of parallelism. The number of elements in the distribution map is the same, although the values differ slightly from those in the previous sort examples.

Clustered Index on Partitioned Table Executed in Parallel

This example creates a clustered index on *orders*, specifying the segment name, *order_seg*.

```
set sort_resources on
alter table orders partition 9
create clustered index order_ix
    on orders (order_id) on order_seg
```

Since the number of available worker processes is 20, this command can use 9 producers and 9 consumers, as shown in the output:

```
The Create Index is done using Parallel Sort
Sort buffer size: 500
Parallel degree: 20
Number of output devices: 9
Number of producer threads: 9
Number of consumer threads: 9
The distribution map contains 8 element(s) for 9
partitions.
Partition Element: 1
```

```
199141
```

```
Partition Element: 2
397543
Partition Element: 3
598758
Partition Element: 4
800484
Partition Element: 5
1010982
Partition Element: 6
1202471
Partition Element: 7
1397664
Partition Element: 8
1594563
Number of sampled records: 8055
```

This distribution map contains 8 elements for the 9 partitions on the table being sorted. The degree of parallelism is 18.

Sort Failure Example

If only 10 worker processes had been available for this command, it could have succeeded using a single producer process to read the entire table. If fewer than 10 worker processes had been available, a warning message would be printed instead of the `sort_resources` output:

```
Msg 1538, Level 17, State 1:
Server 'snipe', Line 1:
Parallel degree 8 is less than required parallel
degree 10 to create clustered index on partition
table. Change the parallel degree to required
parallel degree and retry.
```

Using *sp_sysmon* to Tune Index Creation

You can use the “`begin_sample`” and “`end_sample`” syntax for `sp_sysmon` to provide performance results for individual `create index` commands:

```
sp_sysmon begin_sample  
create index ...  
sp_sysmon end_sample
```

Sections of the report to check include:

- The “Sample Interval”, for the total time taken to create the index
- Cache statistics for the cache used by the table
 - Check the value for “Buffer Grabs” for the 2K and 16K pools to determine the effectiveness of large I/O
 - Check the value “Dirty Buffer Grabs”. If this value is nonzero, set the wash size in the pool higher and/or increase the pool size, using `sp_poolconfig`
- Disk I/O for the disks used by the table and indexes: check the value for “Total Requested I/Os”

Hardware Tuning and Application Maintenance

16

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 will be.

This chapter contains the following sections:

- How Memory Affects Performance 16-1
- How Much Memory to Configure 16-2
- Caches in Adaptive Server 16-3
- The Procedure Cache 16-3
- The Data Cache 16-6
- Named Data Caches 16-11
- Commands to Configure Named Data Caches 16-16
- Cache Replacement Strategies and Policies 16-17
- Configuring and Tuning Named Caches 16-21
- Named Data Cache Recommendations 16-24
- Maintaining Data Cache Performance for Large I/O 16-33
- Speed of Recovery 16-36
- Auditing and Performance 16-38

Chapter 8, "Configuring Memory" in the *System Administration Guide* describes the process of determining 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:

- Not enough total memory is allocated to Adaptive Server.
- Other Adaptive Server configuration options are set too high, resulting in poor allocation of memory.
- 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.

How Much Memory to Configure

Memory is the most important consideration when you are configuring Adaptive Server. Setting the total memory configuration parameter correctly is critical to good system performance.

To optimize the size of memory for your system, a System Administrator calculates the memory required for the operating system and other uses and subtracts this from the total available physical memory. When you start Adaptive Server, it attempts to require the specified amount of memory from the operating system.

If total memory is set too low:

- Adaptive Server may not start.
- If it does start, Adaptive Server may access disk more frequently.

If total memory is set too high, too much memory:

- Adaptive Server may not start.
- If it does start, the operating system page fault rate will rise significantly and the operating system may need to be reconfigured to compensate.

Chapter 8, “Configuring Memory,” in the *System Administration Guide* provides a thorough discussion of:

- How to configure the total amount of memory used by Adaptive Server
- Other configurable parameters that use memory, which affects the amount of memory left for processing queries

When Adaptive Server starts, it allocates memory for the executable and other static memory needs. What remains after all other memory needs have been met is available for the procedure cache and the data cache. Figure 16-1 shows how memory is divided.

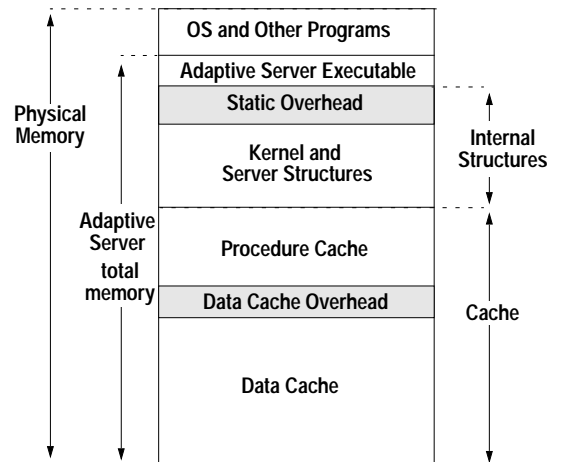


Figure 16-1: How Adaptive Server uses memory

Caches in Adaptive Server

The memory that remains after Adaptive Server has allocated static overhead and kernel and server structures is allocated to:

- The **procedure cache** – used for query plans, stored procedures and triggers.
- The **data cache** – used for all 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.

The split between the procedure cache and the data caches is determined by configuration parameters. A System Administrator can change the distribution of memory available to the data and

procedure caches by changing **procedure cache percent** configuration parameter.

The 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.

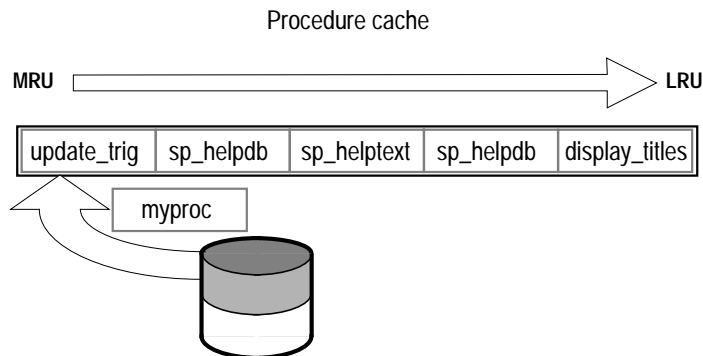


Figure 16-2: The procedure cache

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.

An increase in procedure cache size causes a corresponding decrease in data cache size, as shown in Figure 16-3.

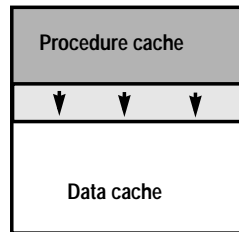


Figure 16-3: Effect of increasing procedure cache size on the data cache

When you first install Adaptive Server, the default procedure cache size is configured as 20 percent of memory that remains after other memory needs have been met. The optimum value for the procedure cache varies from application to application, and it may also vary as usage patterns change throughout the day, week, or month. The configuration parameter to set the size, `procedure cache percent`, is documented in Chapter 11, “Setting Configuration Parameters,” in the *System Administration Guide*.

Getting Information About the Procedure Cache Size

When Adaptive Server is started, the error log states how much procedure cache is available, as shown in Figure 16-4.

```

Maximum number of procedures in cache
      Number of proc buffers allocated: 6632.
      Number of blocks left for proc headers: 7507.
      Procedure cache size, in pages
  
```

Figure 16-4: Procedure cache size messages in the error log

proc buffers

The number of “proc buffers” represents the maximum number of compiled procedural objects that can reside in the procedure cache at one time. In Figure 16-4, no more than 6632 compiled objects can reside in the procedure cache simultaneously.

proc headers

“proc headers” represents number of 2K pages dedicated to the procedure cache. In Figure 16-4, 7507 pages are dedicated to the procedure cache. Each object in cache requires at least 1 page.

Procedure Cache Sizing

How big should the procedure cache be? 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 16-5 make a good starting point.

$$\text{Procedure cache size} = \frac{(\text{Max \# of concurrent users}) * (\text{Size of largest plan}) * 1.25}{1}$$

$$\text{Minimum procedure cache size needed} = \frac{(\# \text{ of main procedures}) * (\text{Average plan size})}{1}$$

Figure 16-5: Formulas for sizing the procedure cache

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 16-5.

Remember, when you increase the size of the procedure cache, you decrease the size of the data cache.

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.

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

```

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 24-81.

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.

The Data Cache

After other memory demands have been satisfied, all remaining space is available in the data cache. 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
- Parts of frequently accessed tables

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 following pages describe the way this single data cache is used. “Named Data Caches” on page 16-11 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.
- 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 “number of index trips” on page 11-26 and “number of oam trips” on page 11-27 of the *System Administration Guide* for more information.
- For queries that scan heaps or tables with clustered indexes or perform nonclustered index scans, Adaptive Server may choose to use 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. A housekeeper task also writes dirty pages to disk when idle time is available between user processes.

Effect of Data Cache on Retrievals

Figure 16-6 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. 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.

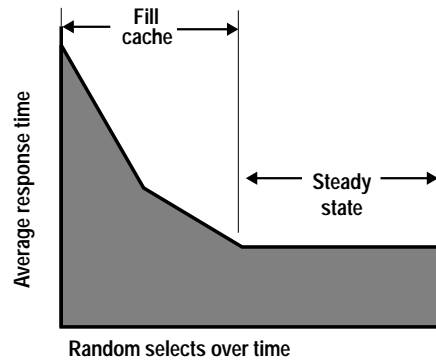


Figure 16-6: Effects of random selects on the data cache

If the cache is smaller than the total number of used pages, 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 need. But caching decreases the likelihood that the maximum delay will be suffered by a particular query—more queries are likely to find at least some of the required pages in cache.

Effect of Data Modifications on the Cache

The behavior of the cache in the presence of update transactions is more complicated than for retrievals. There is still an initial period during which the cache fills. Then, because cache pages are being modified, there is a point at which the cache must begin writing those pages to disk before it can load other pages. Over time, the amount of writing and reading stabilizes, and subsequent transactions have a given probability of requiring a disk read and another probability of causing a disk write. The steady-state period is interrupted by checkpoints, which cause the cache to write all dirty

pages to disk. Figure 16-7 shows how update transactions affect the average response time.

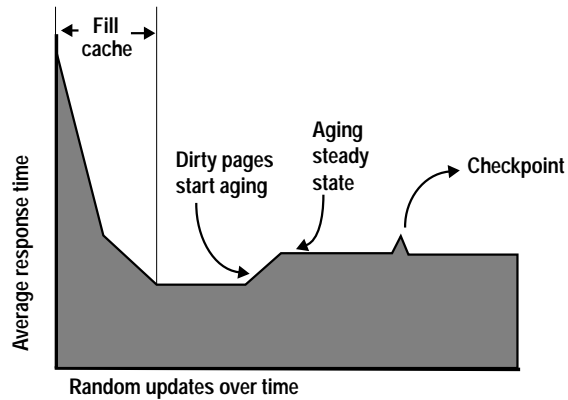


Figure 16-7: Effects of random data modifications on the data cache

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

It is important to 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 such 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, you need to be sure that the mix of queries executed during the tests accesses the database in patterns that are 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`, 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 the formula in Figure 16-8.

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

Figure 16-8: Formula for computing the cache hit ratio

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*

The `sp_sysmon` system procedure 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 24-68.

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 24-74.

Named Data Caches

When you install Adaptive Server, it has a single default data cache with a 2K memory pool. To improve performance, you can split this cache into multiple named data caches and bind databases or database objects to them.

Named data caches are not a substitute for careful query optimization and indexing. In fact, splitting the large default cache into smaller caches and restricting I/O to them can lead to worse performance. For example, if you bind a single table to a cache, and it makes poor use of the space available, no other objects on Adaptive Server can use that memory.

You can configure 4K, 8K, and 16K memory pools 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.

Named Data Caches and Performance

Adding named data caches can improve performance in the following ways:

- 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 is controlled by a separate spinlock, increasing concurrency on systems with multiple CPUs.
- You can configure 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. These caches can be configured 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 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 percent of the queries reference data from the most recent month (about 8 percent of the table), configuring a cache of about 10 percent 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 (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. All processes that create worktables or temporary tables use *tempdb*, so binding it to its own cache keeps its cache use 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.
- You can bind a database’s log to a cache, again reducing contention for cache space and access to the cache.

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.

Large I/Os 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. 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.

Large I/O can increase performance for:

- Queries that table scan—both single-table queries and queries that perform joins
- Queries that scan the leaf level of a nonclustered index

- Queries that use text or image data
- Queries such as `select into` that allocate several pages
- `create index` commands
- Bulk copy operations on heaps—both copy in and copy out
- The `update statistics`, `dbcc checktable`, and `dbcc checkdb` commands

When a cache is configured for 16K I/O and the optimizer chooses 16K I/O for the query plan, Adaptive Server reads an entire extent, eight 2K data pages, when it needs to access a page that is not in cache. Occasionally, 16K I/O cannot be performed. See “When prefetch Specification Is Not Followed” on page 10-12.

Types of Queries That Can Benefit from Large I/O

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 text and image page chain must be bound to a named data cache that has large memory pools, or the default data cache must have large I/O pools. Most of the queries shown here use fetch-and-discard (MRU) replacement strategy. Types of queries that can benefit from large I/O are:

- Queries that scan entire tables, either heap tables or tables with clustered indexes:


```
select title_id, price from titles
select count(*) from authors
      where state = "CA" /* no index on state */
```
- Range queries on tables with clustered indexes:


```
where indexed_colname < value
where indexed_colname > value
where indexed_colname between value1 and value2
where indexed_colname > value1
      and indexed_colname < value2
where indexed_colname like "string%"
```
- Queries that scan the leaf level of a nonclustered index, both matching and nonmatching 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 select *text* or *image* columns:

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

- Join queries where a full scan of the inner table is required:

```
select outer.c1, inner.c3
  from outer, inner
  where outer.c1 = inner.c2
```

If both tables use the same cache, and one of the tables fits completely in cache, that table is chosen as the inner table and loaded into cache with the LRU replacement strategy, using large I/O, if available. The outer table can also benefit from large I/O, but uses the fetch-and-discard (MRU) replacement strategy, so that the pages are read into cache at the wash marker, since the pages for the outer table are needed only once to satisfy the query. Figure 16-9 shows two tables using different caching strategies.

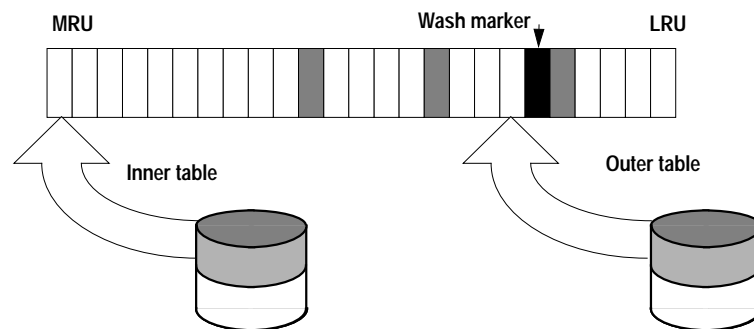


Figure 16-9: Caching strategies joining a large table and a small table

If neither table fits completely in cache, the MRU replacement strategy will be used for both tables, using large I/Os, if they are available in the cache.

- Queries that generate Cartesian products:

```
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.

The Optimizer and Cache Choices

By the time Adaptive Server has optimized a query and needs to access data pages, it:

- Has a good estimate of the number of pages it needs to read for each table.
- Knows the size of the data cache(s) available to the tables and indexes in the query and the I/O size available for the cache(s). The I/O size and cache strategy are incorporated into the query plan.
- Has determined whether the data will be accessed via a table scan, clustered index access, nonclustered index, or another optimizer strategy.
- Has determined which cache strategy to use for each table and index.

The optimizer's knowledge is limited to the single query it is analyzing and to certain statistics about the table and cache. It does not have information about how many other queries are simultaneously using the same data cache, and it 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.

In a few instances, an 8K pool may provide better performance than a 16K pool. For example:

- In some applications with extremely heavy logging, 8K log I/O might perform better than 4K log I/O, but most performance testing has shown 4K log I/O to be optimal.
- If a 16K pool is not being used due to storage fragmentation or because many of the needed pages are already in a 2K pool, an 8K pool might perform better than a 16K pool. For example, if a single page from an extent is in the 2K pool, 7 2K I/Os would be

needed to read the rest of the pages from the extent. With an 8K pool, 1 8K I/O (4 pages) and 3 2K I/Os could be used to read the 7 pages. However, if a 16K pool exists, and a large I/O is denied, Adaptive Server does not subsequently try each successively smaller pool, but immediately performs 2K I/O on the pages needed by the query. You would configure an 8K pool only if a 16K pool was not effective in reducing I/O.

Commands to Configure Named Data Caches

The commands to configure caches and pools are shown in Table 16-1.

Table 16-1: Commands used to configure caches

Command	Function
<code>sp_cacheconfig</code>	Creates or drops named caches and changes the size, cache type, or cache policy. Reports on sizes of caches and pools.
<code>sp_poolconfig</code>	Creates and drops I/O pools and changes their size, wash size, and asynchronous prefetch limit.
<code>sp_bindcache</code>	Binds databases or database objects to a cache.
<code>sp_unbindcache</code>	Unbinds the specified database or database object from a cache.
<code>sp_unbindcache_all</code>	Unbinds all databases and objects bound to a specified cache.
<code>sp_helpcache</code>	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.
<code>sp_sysmon</code>	Reports statistics useful for tuning cache configuration, including cache spinlock contention, cache utilization, and disk I/O patterns.

For a full description of configuring named caches and binding objects to caches, see Chapter 9, “Configuring Data Caches,” in the *System Administration Guide*. Only a System Administrator can configure named caches and bind database objects to them.

For information on `sp_sysmon`, see Chapter 24, “Monitoring Performance with `sp_sysmon`.”

Commands for Tuning Query I/O Strategies and Sizes

You can affect the I/O size and cache strategy for select, delete, and update commands:

- For information about specifying the I/O size, see “Specifying I/O Size in a Query” on page 10-10.
- For information about specifying cache strategy, see “Specifying the Cache Strategy” on page 10-13.

Cache Replacement Strategies and Policies

The Adaptive Server optimizer uses two cache replacement strategies in order 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.

Cache Replacement 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 Figure 3-9 and Figure 3-11 on page 3-16 for illustrations of these strategies.

Cache Replacement 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 sorted order.

The default cache replacement policy is strict replacement. Relaxed replacement policy should be used only when both of the following 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.

Cache Replacement Policy Concepts

Most concepts that apply to strict LRU replacement policy also apply to relaxed replacement policy caches:

- Both policies use a wash marker to determine start I/O on any pages that have been changed in cache. This wash marker is set at a configurable distance from the next buffer in the cache that will be replaced, allowing enough time for dirty pages to be written to disk before the buffer needs to be reused.
- Both policies clearly define which buffer is the next buffer to be reused when a physical I/O needs to take place. In strict LRU caches, this is the buffer at the LRU end of the cache. In relaxed LRU caches, a “victim pointer” indicates the next buffer to be considered for replacement. (A **victim** is the page that is replaced

when a physical read takes place.) If the victim buffer has been recently used, it is not replaced, and the victim pointer moves to the next buffer.

- Both policies try to keep frequently used pages in the cache. With strict replacement policy, this is accomplished by linking pages at the MRU end of the cache chain. With relaxed LRU policy, a status bit in the buffer is set to indicate that the page was recently used and should not be replaced.

Many structures and concepts in strict and relaxed LRU caches are the same, as shown in Figure 16-10.

In strict LRU caches, it is useful to think of pages moving to the head of the MRU/LRU chain after they are used, and of other pages in the cache moving past the wash marker and into the wash area. In contrast, pages in relaxed LRU caches do not move; instead, the wash marker and the victim pointer move each time a buffer needs to be read into the cache.

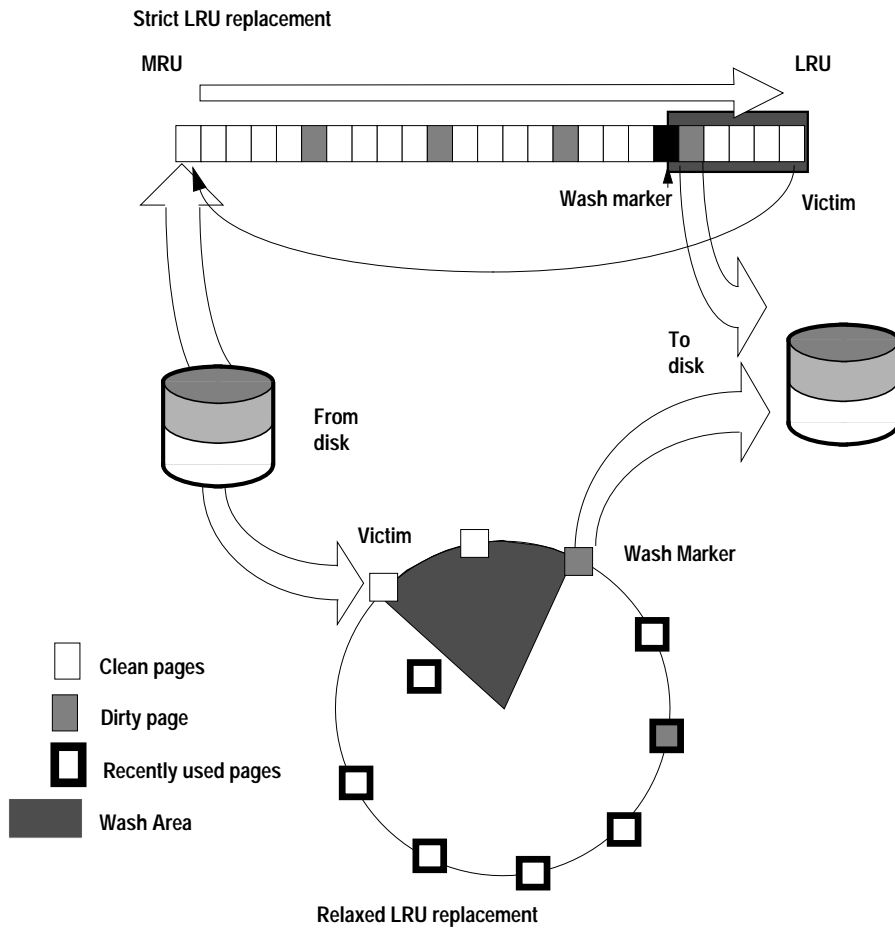


Figure 16-10: Concepts in strict and relaxed LRU caches

Comparing Cache Replacement Policies

Various activities in the data cache have different effects, depending on the cache policy. Table 16-2 compares these activities and their effects.

Table 16-2: Differences between strict and relaxed LRU replacement policy

Action	Strict LRU	Relaxed LRU
Accessing a page already in cache	Buffers are moved or relinked to the top of the MRU chain each time they are accessed in cache.	Buffers are not moved or relinked when they are accessed in the cache. Instead, a bit in the buffer header is set to indicate that this page has been recently used. The recently used bits are turned off at the wash marker.
Choosing the victim buffer for replacement	The page at the LRU is always the victim (the page that is replaced when a physical read takes place).	If the page at the victim pointer has been recently used, it is skipped, and the wash and victim pointers are moved to the next page until a page that has not been recently used is located.
Reaching the wash marker	Any dirty pages are written to disk.	Any dirty pages are written to disk, and the recently used bit is turned off.
Using fetch-and-discard strategy	Fetch-and-discard strategy links buffers into cache at the wash marker.	Fetch-and-discard strategy replaces the victim buffer with the newly read buffer and leaves the recently used bit turned off.
Finding a page in the wash section	If a page from the wash section is accessed by a query, that buffer is linked to the MRU.	The recently used bit is turned on, and the page remains in the wash section. The bit will still be on when this buffer is considered as a victim, so this buffer will not be replaced.
Finding 100% of needed pages in cache, once a stable state is reached.	Each time pages are accessed, they are relinked to the top of the MRU/LRU chain.	The wash marker and victim pointer never move. Once all of the recently used bits are turned on, they remain on.
Checkpoint and the housekeeper	Both checkpoint and the housekeeper write dirty pages to disk.	Checkpoint writes dirty pages to disk, but the housekeeper ignores relaxed LRU caches.

How Relaxed LRU Caches Work

When a buffer needs to be read into a relaxed LRU cache, the recently used status of the victim buffer is checked. The victim is only replaced if the bit is off. The process is as follows:

- If the recently used bit is not set, the page is read into the victim buffer. The victim pointer and wash marker each move to the next buffer. If the buffer at the wash marker is dirty, the disk write is started.
- If the recently used bit is turned on, and buffer is not replaced—it means that this page was accessed while it was in the wash area. the victim pointer and the bit is turned off the wash marker each move to the next buffers in the cache, and status bit for the buffer at the victim buffer is checked.

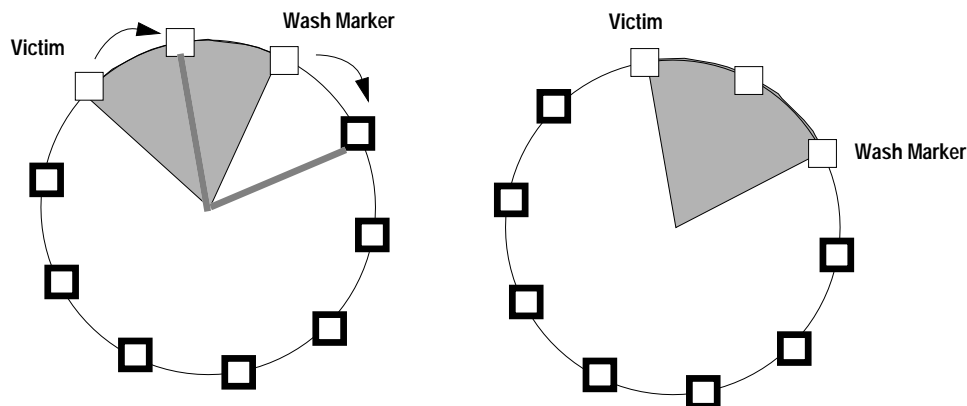


Figure 16-11: Victim and wash pointers moving in an LRU cache

Configuring and 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 percent 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 may experience improved cache hit ratios. The large number of pages often used by queries that perform 16K I/O will 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 used.

In addition to commands such as `showplan` and `statistics io` 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.

Development vs. Production Systems

In an ideal world, you would have access to a development system that duplicates the configuration and activity of your production system. You could tune your cache configuration on the development system and reproduce the perfected configuration on your production server. In reality, most development systems can provide only an approximate test-bed for cache configuration; fine-tuning cache sizes on a development system must be done incrementally in small windows of time when the system can be restarted. Tuning pool sizes and cache bindings is dynamic and, therefore, more flexible, since a re-start of the server is not required.

In a production environment, the possible consequences of misconfiguring caches and pools are significant enough that you should proceed carefully, and only after a thorough analysis of the issues discussed in the following sections. In a test or development environment, you can use `sp_sysmon` or other monitoring tools to run individual queries, particular query mixes, or simulations of your production environment to develop an understanding of how these queries interact in various cache configurations.

When you transfer what you have learned from a test environment to a production environment, remember that the relative size of objects and caches can be critical to the choice of query plans, especially in the case of join queries using the same cache. If your tests involved tables of a few hundred pages, and your production database's tables are much larger, different cache strategies may be chosen for a cache of the same size.

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:

- Configure Adaptive Server with as much total memory as possible. See Chapter 8, “Configuring Memory” in the *System Administration Guide* for more information.
- Once all other configuration parameters that use Adaptive Server memory have been configured, check the size of the default data cache with `sp_cacheconfig` to determine how much space is available.
- Use your performance monitoring tools to establish baseline performance, and to establish your tuning goals.

The process of dividing the cache involves looking at existing objects and applications:

- Evaluate cache needs by analyzing I/O patterns, and evaluate pool needs by analyzing query plans and I/O statistics.
- Configure the easiest choices and biggest wins 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 adding caches is to reduce spinlock contention, moving a few high-I/O objects to separate caches may be sufficient to reduce the spinlock contention and improve 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 24-86 for information about the `sp_sysmon` output showing disk I/O.

Named Data Cache Recommendations

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

- Bind *tempdb* to its own cache, and configure the cache for 16K I/O for use by *select into* queries, if these are used in your applications.
- Bind the logs for your high-use databases to a named data cache. 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 16-27.
- If a table or its index is small and constantly in use, configure a cache just for that object or for a few objects.
- For caches with cache hit ratios of more than 95 percent, configure relaxed LRU cache replacement policy, especially if you are using multiple engines.
- Keep cache sizes and pool sizes proportional to the cache utilization objects and queries:
 - If 75 percent of the work on your server is performed in one database, it should be allocated approximately 75 percent 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 percent of the work in your database can use large I/O, configure about 50 percent of the cache in a 16K memory pool.
- It is better to view the cache as a shared resource than to try to micro-manage 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 percent on a cache, split it into multiple caches. Use *sp_sysmon* periodically during high-usage periods to check for cache contention. See “Spinlock Contention” on page 24-74.

- Set relaxed LRU cache policy on caches with cache hit ratios of more than 95 percent, 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 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 re-read 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 re-reading 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 24-47 for more information.

Configuring for Large Log I/O Size

To check the log I/O size for a database, you can check the server’s error log. The I/O size for each database is printed in the error log when Adaptive Server starts. You can also use the `sp_logiosize` system procedure. 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 cache it is explicitly bound to or the cache used by its database)
- 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.

Basing Data Pool Sizes on Query Plans and I/O

When you choose to divide a cache for tables and/or indexes into pools, try to make this division 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 actually see worse performance. Most of the space in the 2K pool will remain unused, and the 16K pool will experience high turnover. The cache hit ratio will be significantly reduced. The problem will be most severe with join queries that have to repeatedly re-read the inner table from disk.

Making a good choice about pool sizes requires:

- A thorough 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. For example, a query that table scans and performs 800 physical I/Os using a 2K pool should perform about 100 8K I/Os. See “Types of Queries That Can Benefit from Large I/O” on page 16-13 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. Here are some guidelines:

- If you observe that most I/O is occurring in point queries using indexes to access a small number of rows, make the 16K pool relatively small, say about 10 to 20 percent of the cache size.
- If you estimate that a large percentage of the I/Os will use the 16K pool, configure 50 to 75 percent of the cache for 16K I/O. Queries that use 16K I/O include any query that table scans, 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 percent 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 percent 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 especially for join queries where an inner table would use 16K I/O, and the table is repeatedly scanned using 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 10-12.

Configuring Buffer Wash Size

The wash area for each pool in each cache is configurable. If the wash size is set too high, Adaptive Server may perform unnecessary writes. If the wash area is 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. See “Changing the Wash Area for a Memory Pool” on page 9-18 of the *System Administration Guide* for more information.

Choosing Caches for Relaxed LRU Replacement Policy

Relaxed LRU replacement policy can provide performance gains, especially on servers with several engines. Consider adding caches

for specific tables or indexes, as well as configuring existing caches to use relaxed LRU replacement policy.

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 percent, determine whether slightly increasing the size of the cache can provide enough space to completely hold the table or index.

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 for transactions to complete in order 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.

Ideal performance for an operation that performs large I/O while doing a complete table scan is approximately:

$$\text{I/Os} = \frac{\text{Number of pages in table}}{\text{Number of pages per I/O}}$$

For example, if a table has 624 data pages, and the cache is configured for 16K I/O, Adaptive Server reads 8 pages per I/O. Dividing 624 by 8 equals 78 I/Os. If a table scan that performs large I/O performs significantly more I/O than the optimum, you should explore the causes.

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 many other processes have brought pages from the table into the 2K cache. If Adaptive Server is performing 16K I/O and finds that one of the pages it needs to read is 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 255 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, an extent may store both leaf-level pages and pages from higher levels of the index. Regular index access, finding pages by starting from the root index page and following index pointers, always performs 2K I/O. 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. Note that this applies only to nonclustered indexes and their leaf pages, not to clustered index pages and the data pages, which are always stored on separate extents.
- The table storage is fragmented, due to page-chain pointers that cross extent boundaries and allocation pages. Figure 16-12 shows a table that has become fragmented.

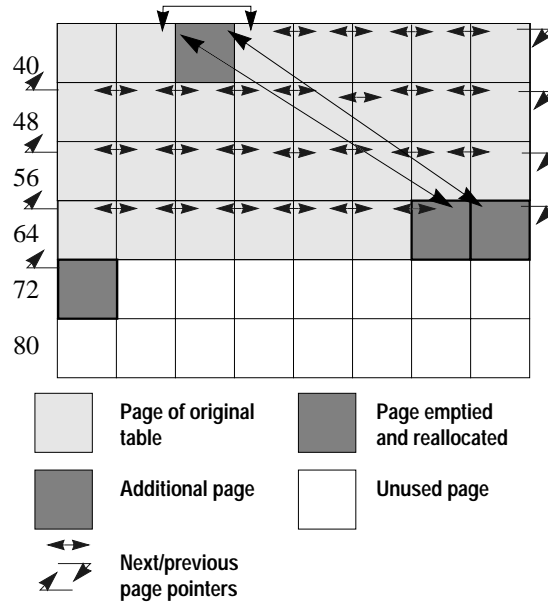


Figure 16-12: Fragmentation on a heap table

The steps that lead to the memory use in the Table 16-12 are as follows:

1. The table is loaded. The gray boxes indicate the original pages of the table.
2. The first additional page, indicated by the first heavily striped box, is allocated for inserts.
3. Deletes cause page 3 of the table, located in extent 40, to be deallocated.
4. Another page is needed, so page 2 is allocated and linked into the page chain, as shown by the lightly striped box.
5. Two more pages are allocated, as shown by the other two heavily striped boxes.

At this point, the table has pages in 5 extents, so 5 large 16K I/Os should be required to read all the pages. Instead, the query performs 7 I/Os. The query reads the pages following the page pointers, so it:

- Performs a 16K I/O to read extent 40, and performs logical I/O on pages 1, 2, and 4–8, skipping page 3.

- Performs physical I/O on extent 48, and, in turn, logical I/O on the each of its pages, and follows the same patterns for extents 56, and 64, in turn
- The second-to-last page in extent 64 points to page 3. It examines page 3, which then points to a page in extent 64.
- The last page in extent 64 points to extent 72.

With a small table, the pages would still be in the data cache, so there would be no extra physical I/O for a small fragmented table like this the one in this example. But when the same kind of fragmentation occurs in large tables, the I/O required rises, especially if a large number of users are performing queries with large I/O that could flush buffers out of the cache.

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:

- “Large I/O Usage” on page 24-79 reports the number of large I/Os performed and denied and provides summary statistics.
- “Large I/O Detail” on page 24-81 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.

Re-Creating Indexes to Eliminate Fragmentation

If I/O for heaps, range queries on clustered indexes, or covering nonclustered indexes exceeds your expected values, use one of the following processes:

- For heaps, either create and then drop a clustered index, or bulk copy the data out, truncate the table, and copy the data in again.
- For clustered indexes, drop and re-create the clustered index. All nonclustered indexes will be re-created automatically.
- For covering nonclustered indexes, drop and re-create the index.

For clustered indexes and nonclustered indexes on tables that will continue to receive updates, using a fillfactor to spread the data slightly should slow fragmentation. This is described in the next section. Fillfactor does not apply to heap tables.

Using Fillfactor for Data Cache Performance

If your table has a clustered index, and queries frequently perform table scans or return large ranges, you should be sure that the table uses a cache that allows 16K I/O, to improve performance.

Tables with clustered indexes can become fragmented by page splits from inserts and expensive direct updates and from the reallocation of pages. Using fillfactor when you create your clustered index can slow this fragmentation.

When you create a clustered index without specifying a fillfactor, the data pages (the leaf level of the clustered index) are completely filled. If you specify a fillfactor of 80, the pages are 80 percent filled. So, for example, instead of 20 rows on a page, there would be only 16, with room for 4 more rows.

Speed of Recovery

As users modify data in Adaptive Server, only the transaction log is written to disk immediately, in order to ensure recoverability. 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 task uses this time to write dirty buffers to disk.
- A user issues a checkpoint command.

The combination of checkpoints, the housekeeper, and writes started at the wash marker has two major 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 5 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 16-3 shows the effects of changing the recovery interval from its current setting on your system.

Table 16-3: 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 and may have to read those pages again very soon. 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 “recovery interval in minutes” on page 11-22 of the *System Administration Guide* for information on setting the recovery interval.

Effects of the Housekeeper Task on Recovery Time

Adaptive Server’s housekeeper 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 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 24-83.

Auditing and Performance

Heavy auditing can affect performance as follows:

- Audit records are written first to a queue in memory and then to the *sybsecurity* database. If the database shares a disk used by other busy databases, it can slow performance.
- If the in-memory audit queue fills up, the user processes that generate audit records sleep. See Figure 16-13 on page 16-39.

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 16-13. 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.

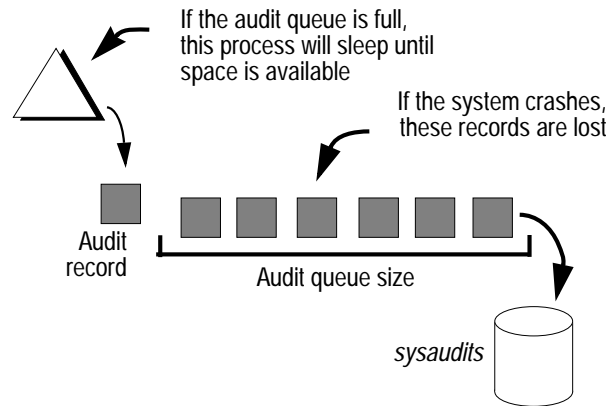


Figure 16-13: Trade-offs in auditing and performance

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.

17 Controlling Physical Data Placement

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

This chapter contains the following sections:

- How Object Placement Can Improve Performance 17-1
- Terminology and Concepts 17-3
- Guidelines for Improving I/O Performance 17-4
- Creating Objects on Segments 17-10
- Partitioning Tables for Performance 17-13
- Space Planning for Partitioned Tables 17-19
- Commands for Partitioning Tables 17-21
- Steps for Partitioning Tables 17-30
- Special Procedures for Difficult Situations 17-37
- Problems When Devices for Partitioned Tables Are Full 17-40
- Maintenance Issues and Partitioned Tables 17-42

How 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. 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 of create database to perform parallel I/O on up to six devices at a time, to gain a significant performance leap for creating multigigabyte 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 24, “Monitoring Performance with *sp_sysmon*.” Pay special attention to the output associated with the discussions in:

- “I/O Device Contention” on page 24-27
- “Heap Tables” on page 24-40
- “Last Page Locks on Heaps” on page 24-61
- “Disk I/O Management” on page 24-86

Adaptive Server Monitor can also help pinpoint problems.

Terminology and Concepts

It is important to 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. Figure 17-1 shows a disk partition initialized as the logical device *userdev1*. See the Adaptive Server installation and configuration guides for information about specific operating system constraints on disk and file usage.

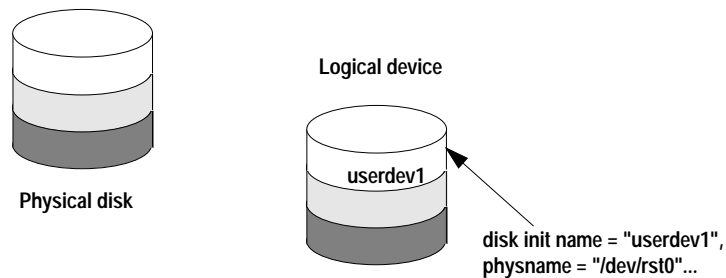


Figure 17-1: Physical and logical disks

- 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 page chain for a table. Partitioning a table splits the table's single page chain so that multiple tasks can access them 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:

- Spread data across disks to avoid I/O contention.
- Isolate server-wide I/O from database I/O.
- Separate data storage and log storage for frequently updated databases.
- Keep random disk I/O away from sequential disk I/O.
- Mirror devices on separate physical disks.

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

Spreading Data Across Disks to Avoid I/O Contention

Spreading data storage across multiple disks and multiple disk controllers avoids bottlenecks:

- 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.

Figure 17-2 shows desirable and undesirable data distribution.

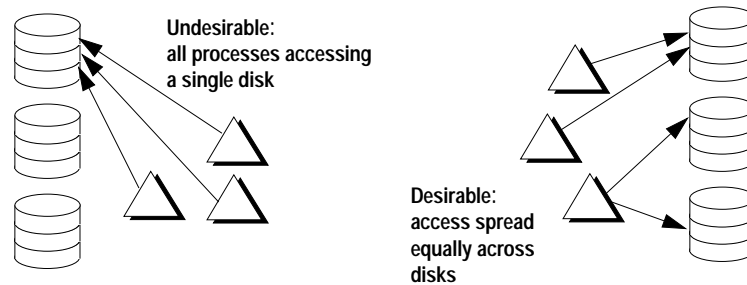


Figure 17-2: Spreading I/O across disks

Avoiding Physical Contention in Parallel Join Queries

The example in Figure 17-3 illustrates a join of two partitioned tables, *orders_tbl* and *stock_tbl*. There are ten worker process 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.

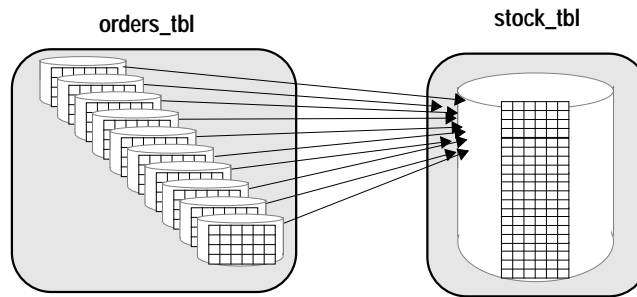


Figure 17-3: Joining tables on different physical devices

Isolating Server-Wide I/O from Database I/O

Place system databases with heavy I/O requirements on separate physical disks and controllers from your application databases, as shown in Figure 17-4.

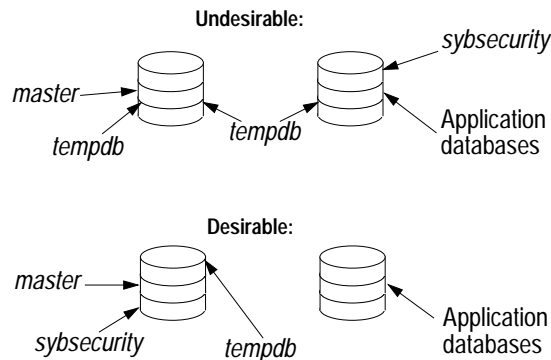


Figure 17-4: Isolating database I/O from server-wide I/O

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.

► **Note**

Using operating system files for user data devices is not recommended on UNIX systems, since these systems buffer I/O in the operating system. Databases placed on operating system files may not be recoverable after a system crash.

See Chapter 19, “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

Placing the transaction log on the same device as the data itself is such a common but dangerous reliability problem that both create database and alter database require the use of the *with override* option if you attempt to put the transaction log on the same device as the data itself.

Placing the log on a separate segment:

- Limits log size, which keeps it from competing with other objects for disk space
- Allows use of threshold management techniques to prevent the log from filling up and to automate transaction log dumps

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

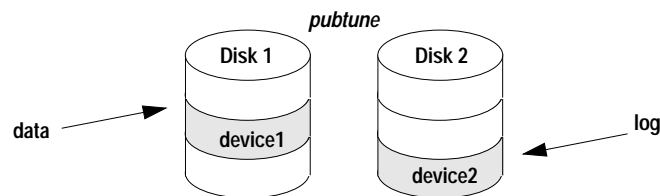


Figure 17-5: Placing log and data on separate physical disks

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. Figure 17-6 shows how separating log and data segments affects log allocations.

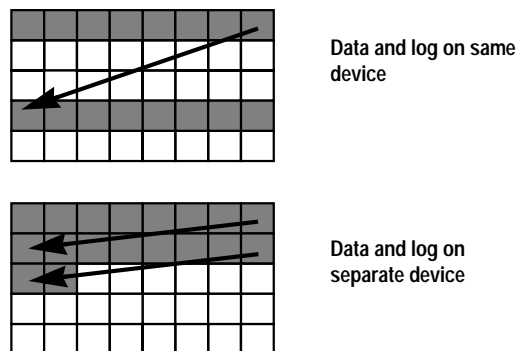


Figure 17-6: Disk I/O for the transaction log

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. This is a serious performance penalty on SMP systems.

If you have created a database without its log on a separate device, see “Moving the Transaction Log to Another Device” on page 15-9 of the *System Administration Guide*.

Mirroring a Device on a Separate Disk

If you mirror data, put the mirror on a separate physical disk from the device that it mirrors, as shown in Figure 17-7. Disk hardware failure often results in whole physical disks being lost or unavailable. Mirroring on separate disks also minimizes the performance impact of mirroring.

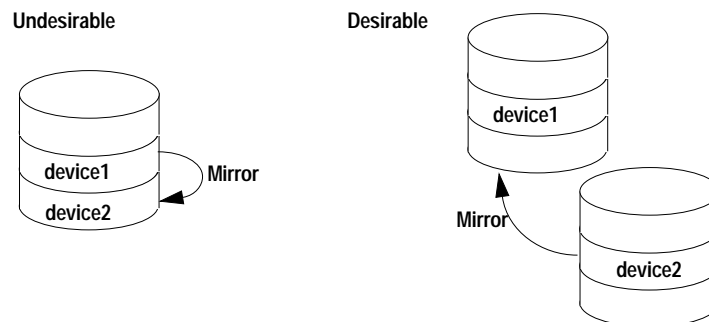


Figure 17-7: Mirroring data to separate physical disks

Device Mirroring Performance Issues

Disk mirroring is a security and high availability feature that allows Adaptive Server to duplicate the contents of an entire database device. See Chapter 7, “Mirroring Database Devices,” in the *System Administration Guide* for more information on mirroring.

Mirroring is not a performance feature. It can slow the time taken to complete disk writes, since writes go to both disks, either serially or simultaneously, as shown in Figure 17-8. Reads always come from

the primary side. Disk mirroring has no effect on the time required to read data.

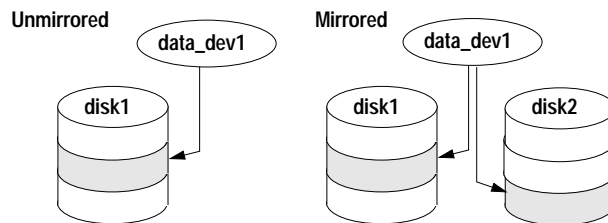


Figure 17-8: Impact of mirroring on write performance

Mirrored devices use one of two modes for disk writes:

- **Noserial** mode can require more time to complete a write than an unmirrored write requires. In noserial mode, both writes are started at the same time, and Adaptive Server waits for both to complete. The time to complete noserial writes is $\max(W_1, W_2)$, the greater of the two I/O times.
- **Serial** mode increases the time required to write data even more than noserial mode. Adaptive Server starts the first write and waits for it to complete before starting the second write. The time required is $W_1 + W_2$, the sum of the two I/O times.

Why Use 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 noserial 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 noserial mode.

Creating Objects on Segments

A segment is best described as a label that points to one or more database devices. Figure 17-9 shows the segment named *segment1*; this segment includes three database devices, *data_dev1*, *data_dev2*, and *data_dev3*. Each database device can use up to 32 segments, including the 3 segments that are created by the system (*system*, *logsegment*, and *default*) when a database is created. Segments label space on one or more logical devices.

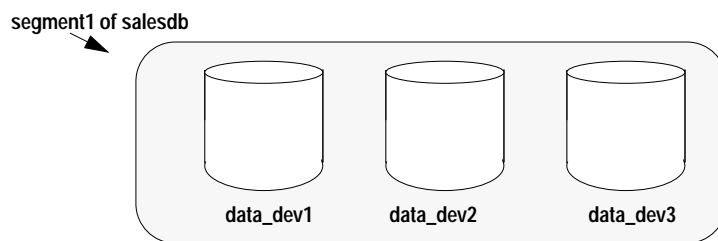


Figure 17-9: Segment labeling a set of disks

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.

Why Use 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. By definition, the leaf level of a clustered index is the table data. 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 cannot separate the clustered index from the data pages.

You can achieve performance improvements by placing nonclustered indexes on a separate segment, as shown in Figure 17-10.

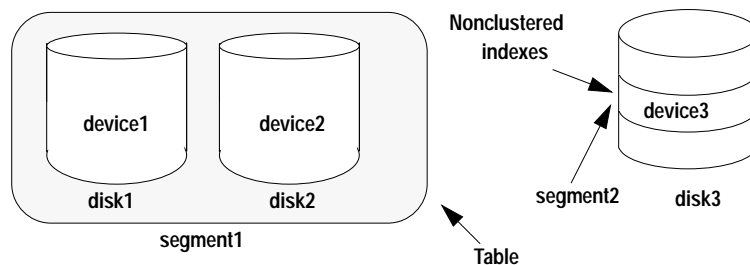


Figure 17-10: Separating a table and its nonclustered indexes

Splitting a Large Table Across Devices

Segments can span multiple devices, so they can be used to spread data across one or more disks, as shown in Figure 17-11. 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.

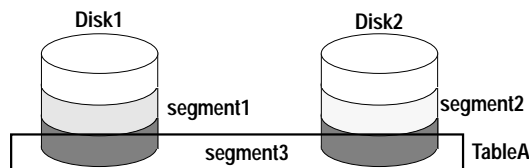


Figure 17-11: Splitting a large table across devices with segments

See “Splitting Tables” on page 17-5 in the *System Administration Guide* for more information.

Moving Text Storage to a Separate Device

When a table includes a *text* or *image* datatype, the table itself stores a pointer to the text or image value. The actual text or image data is stored on a separate linked list of pages. Writing or reading a text value requires at least two disk accesses, one to read or write the pointer and subsequent reads or writes for the text values. If your application frequently reads or writes these values, you can improve performance by placing the text chain on a separate physical device, as shown in Figure 17-12. Isolate text and image chains to disks that are not busy with other application-related table or index access.

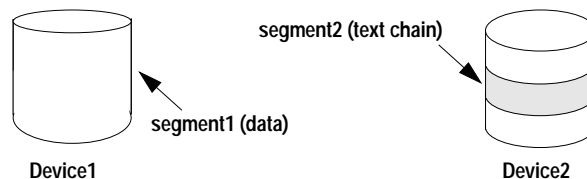


Figure 17-12: Placing the text chain on a separate segment

When you create a table with *text* or *image* columns, Adaptive Server creates a row in *sysindexes* for the object that stores the *text* or *image* 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 *text* or *image* columns in a single table, there is only 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 text columns to a separate segment. See “Placing Text Pages on a Separate Device” on page 17-13 for more information.

Partitioning Tables for Performance

Partitioning a table creates multiple page chains for the table. A **partition** is another term for a page chain. The reasons for partitioning a table are as follows:

- Partitioning adds a degree of parallelism to systems configured to perform parallel query processing. 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 are encountering 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 13-18 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 appear to have a single page chain 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 Chapter 18, “Checking Database Consistency,” in the *System Administration Guide* for information about `dbcc`.
- The `sp_helpartition` system procedure lists information about a table’s partitions.
- `showplan` output displays messages indicating the number of worker processes uses 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

If tables are partitioned, parallel query processing can potentially produce dramatic improvements in query performance. Partitions increase the number of page chains that can be accessed simultaneously 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 14, “Parallel Query Optimization,” 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. Dropping and re-creating the index is required to redistribute the data.

Improving Insert Performance with Partitions

All insert commands on a heap table attempt to insert the rows on the last page of the table's page chain. If multiple users insert data simultaneously, each new insert transaction must wait for the previous transaction to complete in order to proceed. Partitioning a heap table improves the performance of concurrent inserts by reducing contention for the last page of a page chain.

Page Contention During Inserts

By default, Adaptive Server stores a table's data in one double-linked set of pages called a **page chain**. If the table does not have a clustered index, Adaptive Server makes all inserts to the table in the last page of the page chain. If the current last page becomes full, Adaptive Server allocates and links a new last page.

The single-page chain model works well for tables that have modest insert activity. However, as multiple transactions attempt to insert data into the table at the same time, performance problems can occur. Only one transaction at a time can obtain an exclusive lock on the last page, so other concurrent insert transactions wait, or block.

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.

Figure 17-13 shows an example of insert activity in a heap table with three partitions.

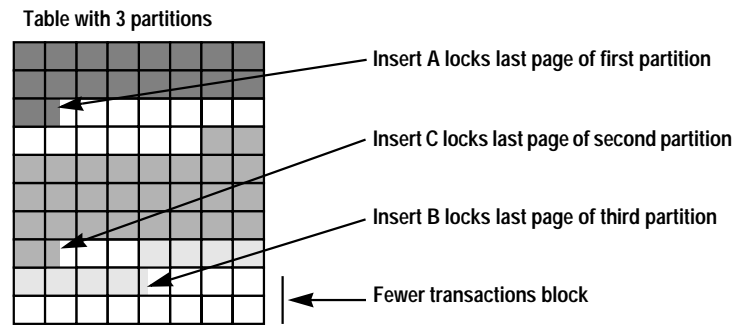


Figure 17-13: Addressing page contention with partitions

Selecting Heap Tables to Partition

Heap tables that have large amounts of concurrent insert activity will benefit from partitioning. 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 24-58.

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 17-22 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 Chapter 11, “Setting Configuration

Parameters,” in 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 17-1 illustrates how Adaptive Server assigns 5 partitions to 3, 5, and 12 devices, respectively.

Table 17-1: 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* or *image* datatypes. However, the *text* and *image* columns themselves are not partitioned—they remain on a single page chain. See “text and image Datatypes” in the *Adaptive Server Reference Manual* for more information about these datatypes.

► Note

Table 17-1 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.

Effects of Partitioning on System Tables

For an unpartitioned table with no clustered index, Adaptive Server stores a pointer to the last page of the page chain in the *root* column of the *sysindexes* row for that table. (The *indid* value for such a row is 0.) When you partition a heap table, the *root* value for that table becomes obsolete. Adaptive Server inserts a row into the *syspartitions* table for each partition and allocates a control page and a first page for each partition. Each row in *syspartitions* identifies a unique partition, along with the location of its control page, first page, and other status information. A partition's control page functions like the *sysindexes.root* value for an unpartitioned table—it keeps track of the last page in the page chain.

Partitioning or unpartitioning a table does not affect the *sysindexes* rows for that table's nonclustered indexes. (The *indid* values for these rows are greater than 1.) *root* values for the table's nonclustered indexes and for partitioned tables with clustered indexes, still point to the root page of each index, since the indexes themselves are not partitioned.

See *sysindexes* and *syspartitions* in the *Adaptive Server Reference Manual* for more details about these system tables.

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 percent of the space occupied by the table to drop and re-create the index

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 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 percent of the space occupied by the table.

If that much space is not available, alternative methods take several steps, and require copying data out and back into the table to achieve load balance and to reduce fragmentation. You need to estimate the frequency at which you will need to perform partitioned table maintenance, the cost in System Administrator time and the time that it might require tables in your system to be unavailable. Compare this to the cost of adding disk drives. In most cases, the investment in additional hardware is the cheapest solution.

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 17-30 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 percent of the space in the segment. The clustered index tree requires about 20 percent 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, a process 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 17-30.
- 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.

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 percent 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.
- The performance of large I/Os and asynchronous prefetch will almost certainly suffer because the page chain crosses extents and

allocation units. Large I/Os 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 percent of the space occupied by the table. If the space is not available, the 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 17-30.

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 to be created.

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.

► **Note**

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.

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

```
alter table table_name unpartition
```

where *table_name* is the name of the partitioned table.

Unpartitioning a table joins the previous and next pointers of the multiple partitions to create a single page chain. It does not change the location of the data.

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` (see "alter table...unpartition Syntax" on page 17-22). Then invoke the `alter table...partition` clause, 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.

The system procedure `sp_helppartition 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 was used to create the existing clustered index.

Creating a clustered index on a partitioned table requires a parallel sort. Check the following configuration parameters and set options and set them 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 13-10. See Chapter 15, "Parallel Sorting," for more 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 *drop index* and *create clustered index*

If the index on the table was created with `create index`, follow these steps:

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 23-4 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. Drop triggers 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:1 in bigfile -F900001 -L1000000
```

If you have space to split the file into multiple files, copying from separate files is usually much faster than using the first row and last row command line arguments.

Parallel Copy and Locks

Starting many current parallel *bcp* sessions may cause Adaptive Server to run out of locks. Either increase the number of locks configuration parameter or use `-bN` to set a batch size for each *bcp* session.

When you copy in to a table, *bcp* acquires the following locks:

- An exclusive intent lock on the table
- An exclusive page lock on each data page, and exclusive lock on index pages, if any indexes exist

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
- Use the `-b batchsize bcp` flag to copy smaller batches (If you do not use the `-b` flag, the entire copy operations is treated as a single batch.)

You can estimate the number of locks needed with this formula:

```
number_of_batches *  
  (rows_per_batch / (2016/row_length))
```

To see the row length for a table, you can use this query:

```
select maxlen  
  from sysindexes  
  where id = object_id("tablename")  
         and (indid = 0 or indid = 1)
```

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.

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.

The values returned by **sp_helppartition** under the *ptn_data_pages* heading may be inaccurate in a few cases. For a complete description of **sp_helppartition**, see the *Adaptive Server Reference Manual*.

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 command 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 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 17-40 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 Chapter 18, “Checking Database Consistency,” in 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 determine 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 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

It is important to 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 Chapter 17, "Creating and Using Segments," in 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. The major possibilities are:

- 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

Backing Up the Database After Partitioning Tables

Each of the following steps ends with "Dump the database". Using fast bulk copy and creating indexes in parallel both make minimally logged changes to the database, and require a full database dump. Information about the partitions on a table is stored in the `syspartitions` table in each database.

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*.

The Table Does Not Exist

The steps to create a new partitioned table are:

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 17-10.
2. Partition the table, with one partition for each physical device in the segment. See “alter table...partition Syntax” on page 17-21.

► **Note**

If the input data file is not in clustered key order, and the table will occupy more than 40 percent of the space on the segment, and you need a clustered index, see “Special Procedures for Difficult Situations” on page 17-37.

3. Copy the data into the table using parallel bulk copy. See “Using Parallel bcp to Copy Data into Partitions” on page 17-25 for examples using bcp.
4. If you do not need a clustered index, use *sp_helppartition* to verify that the data is distributed evenly on the partitions. See “Getting Information About Partitions” on page 17-26.

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 was 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 builds 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 17-24). To create a clustered index with referential integrity constraints, use *alter table...add constraint* (see “Using Constraints and alter table” on page 17-24.)

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 percent 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 17-34.

5. Create any nonclustered indexes.
6. Dump the database.

The 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 17-22.
2. Partition the table, matching the number of devices on the target segment. See “alter table...partition Syntax” on page 17-21.
3. If a clustered index exists, drop the index. Depending on how your index was created, use either `drop index` (see “Using drop index and create clustered index” on page 17-24) or `alter table...drop constraint` (see “Using Constraints and alter table” on page 17-24.)
4. Create or re-create the index with the `on segment_name` clause. When the segment name is different from the current segment where the table is stored, creating the 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 17-23.
5. If you do not need the clustered index, you can drop it at this point.
6. Dump the database.

The 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 percent 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 percent 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 17-23 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 the space is located 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 17-22.
3. Truncate the table with `truncate table`.
4. Drop the clustered index using `drop table` or `alter table...drop constraint`. See “Distributing Data Evenly Across Partitions” on page 17-23. Also, drop nonclustered indexes, to avoid deadlocking during the parallel bulk copy sessions.
5. Repartition the table. See “alter table...partition Syntax” on page 17-21.
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 17-25.

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 17-22.
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 17-21.
6. Copy the data in using parallel bulk copy. See “Using Parallel bcp to Copy Data into Partitions” on page 17-25.
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

If you want 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 percent 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 17-22.
3. Truncate the table with `truncate table`.

4. Drop nonclustered indexes, to avoid deadlocking during the parallel bulk copy in sessions.
5. Re-partition the table. See “alter table...partition Syntax” on page 17-21.
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 17-25.
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

If you need to add a device to a segment, follow these steps:

1. Check the amount of free space available on the devices in the segment with *sp_helpsegment*. If space on any device is extremely low, see “Problems When Devices for Partitioned Tables Are Full” on page 17-40. 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 17-22.
5. Repartition the table, specifying the new number of devices in the segment. See “alter table...partition Syntax” on page 17-21.
6. If a clustered index exists, drop it and re-create it. Do not use the *sorted_data* option, so that the sort and data redistribution will be performed. See “Distributing Data Evenly Across Partitions” on page 17-23.
7. Dump the database.

Special Procedures for Difficult Situations

These techniques are more complex than those presented earlier in the chapter.

A Technique for Clustered Indexes on Large Tables

If you need to create a clustered index on a table that will fill more than 40 to 45 percent 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 17-22.
3. Truncate the table.
4. Repartition the table. See “alter table...partition Syntax” on page 17-21.
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 17-24) or **alter table...drop constraint** (see “Using Constraints and alter table” on page 17-24).
6. Use parallel bulk copy to copy in enough of the data to fill approximately 40 percent of the segment. Distribute the data evenly on the partitions. This must be a representative sample of the values in the key column(s) of the clustered index. Copying in 40 percent of the data is much more likely to yield good results than smaller amounts of data, and this portion of the bulk copy can be performed in parallel; the second bulk copy operation must be nonparallel. See “Using Parallel bcp to Copy Data into Partitions” on page 17-25.
7. Create the clustered index on the segment. The data will not be in sorted order, so do not use the **sorted_data** clause.
8. Use nonparallel **bcp**, in a single session, to copy in the rest of the data.
9. Check the distribution of data pages on partitions with **sp_helppartition**, and the distribution of pages on the segment with **sp_helpsegment**.

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 will 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.

A Complex Alternative for Clustered Indexes

This set of steps may be useful when:

- The table data occupies more than 40 to 45 percent 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 “A Technique for Clustered Indexes on Large Tables” on page 17-37.

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 17-24 or “Using Constraints and alter table” on page 17-24.
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 make careful note of 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 delimiters 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 17-22.
7. Truncate the table.
8. Repartition the table. See “alter table...partition Syntax” on page 17-21.
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 these rows 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 will 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 existence of the clustered index forces the pages to the correct partition.

13. Check the balance of data pages on the partitions with `sp_helppartition` and the balance of pages on the segments with `sp_helpsegment`.
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. Since the entire table must be copied by a single, slow bulk copy, it is likely to be the slowest method. Also, the existence of the clustered index is likely to lead to page splitting on the data pages, 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 creating an index cannot copy data to that physical device. If a physical device is almost completely full, re-creating the clustered index will 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 17-14 and Figure 17-15 show one such situation.

Devices 2 and 3 are completely full, as shown in Figure 17-14.

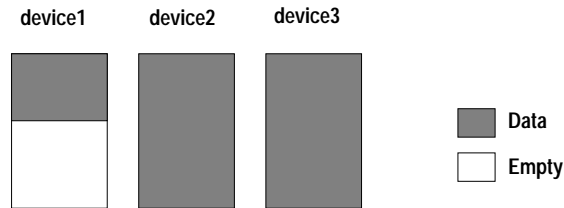


Figure 17-14: A table with 3 partitions on 3 devices

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 has two partitions, and each is 100% full

Figure 17-15 shows these results.

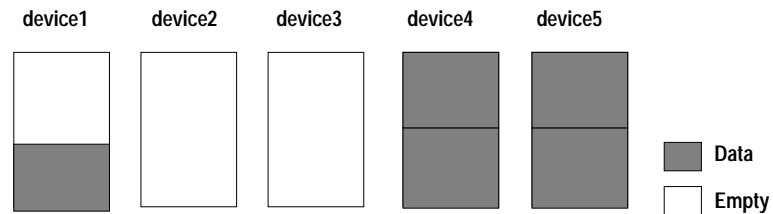


Figure 17-15: Devices and partitions after `create index`

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 will not balance data across devices. Instead, the device that is nearly full will store a small portion of the partition, and the other space allocations for the

partition will steal extents on other devices. Figure 17-16 shows a table with nearly full data devices.

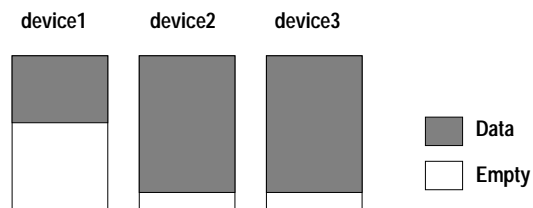


Figure 17-16: Partitions almost completely fill the devices

After adding devices and re-creating the clustered index, the result might be similar to the results shown in Figure 17-17.

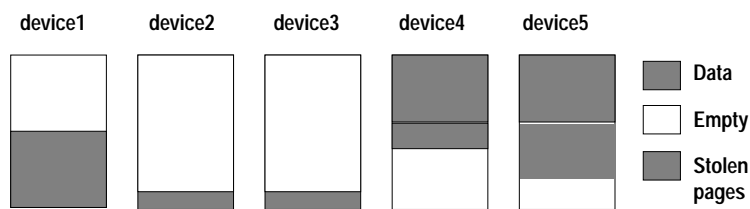


Figure 17-17: Extent stealing and unbalanced data distribution

Once the partitions on *device2* and *device3* used the small amount of space available, they started 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 will 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.

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 are:

- Tables that are read-only or that experience very few updates need little care. 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 will not develop skewed distribution of pages. If the data modifications lead to space fragmentation and partially filled data pages, re-creating the clustered index may be required periodically.
- 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:

- Check the balance on partitions, with `sp_helppartition`. If some partitions are significantly larger or smaller than the average, re-create the clustered index to redistribute data.
- Check the balance of space on underlying disks, with `sp_helpsegment`.
- If you re-create the clustered index to redistribute data for parallel query performance, check for devices that are nearing 50 percent full. Adding disks early, before devices become too full, avoids the complicated procedures described earlier in this chapter.
- Check the space available on each device, with `sp_helpsegment` or `sp_helpdb`. `sp_helpsegment` shows free pages; `sp_helpdb` shows free kilobytes.

In addition, run `update partition statistics`, if partitioned tables undergo the types of activities described in “Updating Partition Statistics” on page 17-29.

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.

18

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.

This chapter contains the following sections:

- How Asynchronous Prefetch Improves Performance 1
- When Prefetch Is Automatically Disabled 18-7
- Tuning Goals for Asynchronous Prefetch 18-10
- Asynchronous Prefetch and Other Performance Features 18-11
- Special Settings for Asynchronous Prefetch Limits 18-13
- Maintenance Activities for High Prefetch Performance 18-14
- Performance Monitoring and Asynchronous Prefetch 18-15

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 in the table's page chain. 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 stalling 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 onto and off of 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 does so 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 3-8.

The 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 up to 24 transactions ahead of the current transaction.

► **Note**

Recovery uses only the 2K pool in the default data cache. See "Setting Asynchronous Prefetch Limits for Recovery" on page 18-13 for more information.

The 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 18-8.

The 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 those pages.

The 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, follow the complete page chains for the objects they check and validate those pages against 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 18-1 summarizes the minimum and maximum sizes for different type of asynchronous prefetch usage.

Table 18-1: 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 (at 2K, maximum is 255; 256 minus the allocation page). • 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 2K pool in the default data cache
	Scanning the transaction log	Maximum is all pages on an allocation unit belonging to the log (255 at 2K)
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 unaccessed 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. Also see “max async i/os per engine” on page 11-78, and “max async i/os per server” on page 11-78 in the *System Administration Guide*.

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 “disk i/o structures” on page 11-36 of 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 24-71.

Unnecessary Reads

Asynchronous prefetch tries to avoid unnecessary physical reads. During recovery and during nonclustered index scans, look-ahead sets are very 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
- 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 can develop kinks. Figure 18-1 shows an example of a kinked page chain between extents in two allocation units.

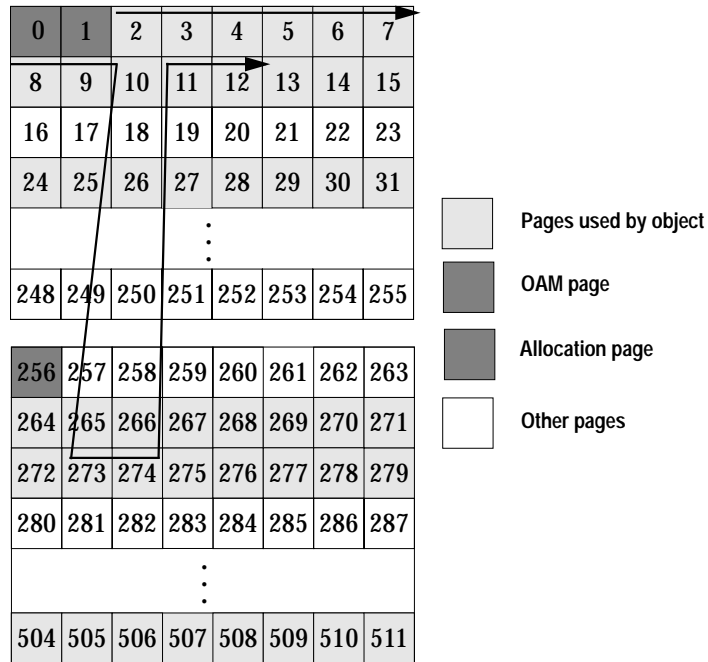


Figure 18-1: A kink in a page chain crossing allocation units

In Figure 18-1, when a sequential 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 9, 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 prefetched 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 2. 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 high 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 Asynchronous Prefetch Limits for Recovery" on page 18-13 and "Setting Asynchronous Prefetch Limits for `dbcc`" on page 18-14 for more information.

Commands to Configure Asynchronous Prefetch

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 Adaptive Server or upgrade to release 11.5, the default value for `global async prefetch`

limit is 10 (percent.) For more information, see “global async prefetch limit” on page 11-25 of the *System Administration Guide*.

- A per-pool override, set with the system procedure `sp_poolconfig`. To see the limits set for each pool, use `sp_cacheconfig`. For more information, see “Changing the Asynchronous Prefetch Limit for a Pool” on page 9-21 of the *System Administration Guide*.

Changing asynchronous prefetch limits takes effect immediately, and does not require a restart of Adaptive Server. Both the global and per-pool limits can also be configured in the configuration file.

Asynchronous Prefetch and Other Performance Features

This section covers the interaction of asynchronous prefetch with other Adaptive Server performance features.

Large I/O and Asynchronous Prefetch

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 (248 2K pages)
- 7 2K I/Os, for the pages that share an extent with the allocation page.

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 and Asynchronous Prefetch

When a scan uses MRU replacement policy, buffers are handled in a special manner when they are read into the cache by asynchronous prefetch. First, pages are linked at the MRU end of the chain, rather than at the wash marker. When the query accesses the page, the buffers are relinked into the pool at the wash marker. This strategy helps to avoid cases where heavy use of a cache flushes prefetched buffers linked at the wash marker before 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, Large I/Os, and Asynchronous Prefetch

The demand on 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 Scans

Hash-based scans 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 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 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 Scans

Partition 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 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 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 Asynchronous Prefetch 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 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 Asynchronous Prefetch 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 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 page chain kinks. Once more than 10 to 20 percent of the original rows in a table have been modified, you should check to 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 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 Chapter 7, “Indexes and I/O Statistics,” for more information.

`sp_sysmon` for release 11.5 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 24-65 and “Disk I/O Management” on page 24-86 for more information.

19

tempdb Performance Issues

This chapter discusses the performance issues associated with using the *tempdb* database. *tempdb* is used by all users of Adaptive Server. Anyone can create objects in *tempdb*. Many processes use it silently. It is a server-wide resource that is used primarily for:

- Internal processing of sorts, creating worktables, reformatting, and so on
- 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.

This chapter contains the following sections:

- How *tempdb* Affects Performance 19-1
- Types and Uses of Temporary Tables 19-2
- Initial Allocation of *tempdb* 19-4
- Sizing *tempdb* 19-4
- Placing *tempdb* 19-8
- Dropping the master Device from *tempdb* Segments 19-8
- Binding *tempdb* to Its Own Cache 19-10
- Temporary Tables and Locking 19-10
- Minimizing Logging in *tempdb* 19-11
- Optimizing Temporary Tables 19-11

How *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(coll)
```

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(coll)
```

Worktables

Worktables are automatically created in *tempdb* by Adaptive Server for 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 19-1. 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.

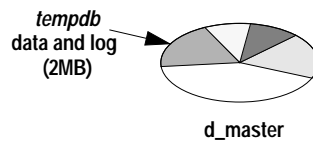


Figure 19-1: tempdb default allocation

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, 1995  select into/bulkcopy

device_frag  size      usage          free kbytes
-----
master       2.0 MB   data and log  1248

```

Sizing *tempdb*

tempdb needs to be big enough to handle the following processes for every concurrent Adaptive Server user:

- Internal sorts
- Other internal worktables that are created for **distinct**, **group by**, and **order by**, for reformatting, and for the OR strategy
- 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 4 tables
- Queries that exceed the 16-table join limit
- 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 in order to do aggregate processing

Information for Sizing *tempdb*

To estimate the correct size for *tempdb*, you need the following information:

- Maximum number of concurrent user processes (an application may require more than one process)
- Size of sorts for queries with **order by** clauses that are not supported by an index
- Size of worktables for reformatting, **group by**, **distinct**, and the **OR** strategy (but not for sorts)
- Number of steps in the query plans for reformatting, **group by**, and so on, which indicates the number of temporary tables created
- Number of local and remote stored procedures and/or user sessions that create temporary tables and indexes
- Size of temporary tables and indexes, as reported by `statistics io` or `sp_spaceused`
- Number of temporary tables and indexes created per stored procedure

Estimating Table Sizes in *tempdb*

To find the size of temporary tables, you can use the `sp_spaceused` system procedure from *tempdb*.

Getting an estimate of the size of worktables is more difficult. One approach is to take the query that requires a worktable and create a temporary table with all of the result rows and columns that are placed in the worktable. For example, this query requires a worktable and sort to eliminate duplicates:

```
select distinct city
from authors
```

The worktable created for this query includes just one column, *city*, and a row for each row in the *authors* table. You could use this query to create a temporary table, and then use `sp_spaceused` on the *#tempcity* table:

```
select city
into #tempcity
from authors
```

If a query uses `group by` or `order by` on columns that are not in the select list, you need to add those columns to the select list in order to get an accurate size estimate.

See Chapter 6, “Determining or Estimating the Sizes of Tables and Indexes,” for more information about gathering size data.

tempdb Sizing Formula

The 25 percent padding in the calculations below covers other undocumented server uses of *tempdb* and covers the errors in our estimates.

1. Compute the size required for usual processing:

Sorts	Users * Sort_size		_____
Other	Users * Worktable_size	+	_____
Subtotal		=	_____
		*	_____
			# of query plan steps
Total for usual processing		=	_____

2. Compute the size required for temporary tables and indexes:

Temporary tables	Procs * Table_size * Table_number		_____
Indexes	Procs * Index_size * Index_number	+	_____
Total for temporary objects		=	_____

3. Add the two totals, and add 25 percent for padding:

Processing			_____
Temp tables	+		_____
Estimate	=		_____
	*	1.25	
Final estimate	=		_____

Example of *tempdb* Sizing

1. Processing requirements:

Sorts	55 users * 15 pages =	825 pages
Other	55 users * 9 pages =	495 pages
Subtotal		_____
		= 1320 pages
		* 3 steps
Total for usual processing		_____
		3960 pages, or 8.2MB

2. Temporary table/index requirements:

Temporary tables	190 procs * 10 pages * 4 tables =	7600 pages
Indexes	190 procs * 2 pages * 5 indexes =	190 pages
Total for temporary objects		_____
		7790 pages, or 16MB

3. Add the two totals, and add 25 percent for padding:

Processing		8.2MB
Temp tables	+	16MB
Estimate	=	24.2MB
	*	*1.25
Final estimate	=	30MB

Placing *tempdb*

Keep *tempdb* on separate physical disks from your critical application databases at all costs. 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.

These are the principles to apply when deciding where to place *tempdb*. Note that the pages in *tempdb* should be as contiguous as possible because of its dynamic nature.

- Expand *tempdb* on the same device as the *master* database. If the original logical device is completely full, you can initialize another database (logical) device on the same physical device, provided that space is available. This choice does not help improve performance by spreading I/O across devices.
- Expand *tempdb* on another device, but not one that is used by a critical application. This option can help improve performance.
- Remember that logical devices are mirrored, but not databases. 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.
- Drop the master device from the *default* and *logsegment* segments.

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* are not created wholly or partially 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. The *default* or *logsegment* segment must include at least one database device. For example:

```
alter database tempdb on tune3 = 20
```

2. Issue a use *tempdb* command, and then drop the master device from the segment:

```
sp_dropsegment "default", tempdb, master
sp_dropsegment logsegment, tempdb, master
```

3. To verify that the *default* segment no longer includes the master device, issue the 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 19-2, you can take advantage of parallel query performance for some temporary tables or worktables.

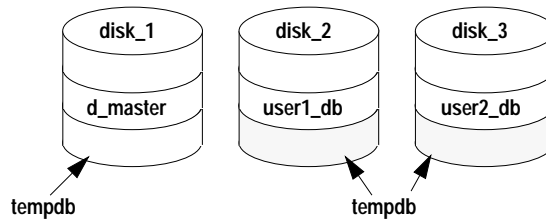


Figure 19-2: tempdb spanning disks

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

Commands for Cache Binding

Use the `sp_cacheconfig` and `sp_poolconfig` commands 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. For instructions on configuring named caches and pools, see “Configuring Data Caches” on page 9-5 of 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

Locking in *tempdb* can be caused by creating or dropping temporary tables and their indexes.

When users create tables in *tempdb*, information about the tables must be stored in system tables such as *sysobjects*, *syscolumns*, and *sysindexes*. Updates to these tables require a table lock. 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

Minimizing Logging 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.

Minimizing Logging 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, since the work of creating the table has not been performed at the time the query is optimized, as shown in Figure 19-3. This applies to both temporary tables and regular user tables.

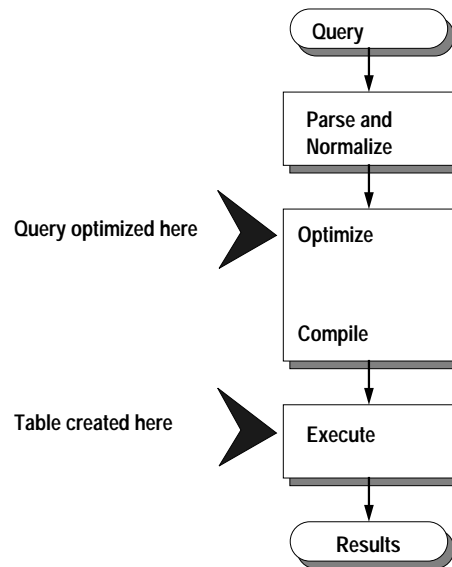


Figure 19-3: Optimizing and creating temporary tables

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 uses 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 index must exist at the time 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 statistics page must exist. If you create the temporary table and create the index on an empty table, Adaptive Server does not create a statistics page. If you then insert data rows, the optimizer has no statistics.
- 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.

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 first procedure calls the second one, the optimizer can determine the size of the table:

```
create proc base__proc
as
    select *
        into #huge_result
        from ...
    exec select_proc

create proc select_proc
as
    select *
        from tab, #huge_result where ...
```

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.

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*.

20

Networks and Performance

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

This chapter contains the following sections:

- Why Study the Network? 20-1
- Potential Network-Based Performance Problems 20-1
- How Adaptive Server Uses the Network 20-3
- Changing Network Packet Sizes 20-3
- Techniques for Reducing Network Traffic 20-6
- Impact of Other Server Activities 20-8
- Guidelines for Improving Network Performance 20-9

Why Study the Network?

You should work with your network administrator to discover potential network problems if:

- 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 Network-Based 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 About Networks and 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 24, “Monitoring Performance with *sp_sysmon*.” Pay special attention to the output in “Network I/O Management” on page 24-92.

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 all Sybase products.

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.

Chapter 11, “Setting Configuration Parameters,” in 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 vs. 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 steal that space. Connections that use the default network packet size always have three buffers reserved for the connection.

Connections that request large packet sizes must 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 that needs to be sent.

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

- Bulk copy
- readtext and writetext commands
- Large select statements

Point of Diminishing Returns

There is always a point at which increasing the packet size will not improve performance, and in fact it may decrease performance, because the packets are not always full. Figure 20-1 shows that increasing the packet size past an optimal setting can increase transfer time. Although there are analytical methods for predicting that point, it is more common to vary the size experimentally and plot the results. If such experiments are conducted over a period of time and conditions, a packet size that works well for a lot of processes can be determined. However, since the packet size can be customized for every connection, specific experiments for specific processes can be beneficial.

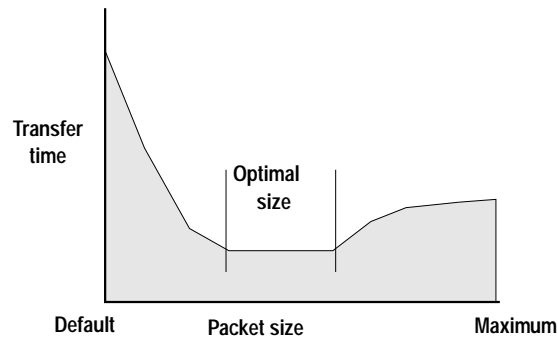


Figure 20-1: Packet sizes and performance

The curve 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.

Client Commands for Larger Packet Sizes

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.

For `isql` and `bcp`, the command line arguments are as follows:

UNIX, Windows NT, and OS/2

```
isql -Asize
```

```
bcp -Asize
```

Novell NetWare

```
load isql -Asize
```

```
load bcp -Asize
```

For Open Client™ Client-Library, use:

```
ct_con_prop(connection, CS_SET, CSPACKETSIZE,
$packetsize (sizeof(packetsize), NULL)
```

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.

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.

Techniques for Reducing Network Traffic

Server-Based Techniques for Reducing 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 to Adaptive Server, converting them to use stored procedures can reduce network traffic.

Clients should request only the rows and columns they need.

Using Stored Procedures to Reduce Network Traffic

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.

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, as shown in Figure 20-2. In many cases, this can also reduce the disk I/O load.

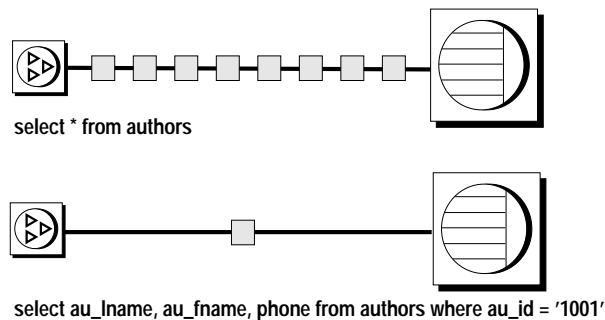


Figure 20-2: Reducing network traffic by filtering data at the server

Fill Up Packets When Using Cursors

Open Client Client-Library applications that use cursors can request multiple rows for each fetch command:

`ct_cursor (CT_CURSOR_ROWS)`

To fetch multiple rows in `isql`, use the `set cursor rows` option.

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 20-1 shows the characteristics of some network types.

Table 20-1: 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 a problem before database users start complaining to their System Administrator. Cooperate with your local network managers and be prepared to provide them with your predicted or actual network requirements when they are considering the addition of resources. Also, keep an eye on the network and try to anticipate problems that result from newly added equipment or application requirements. Remember, network problems affect all the database clients.

Impact of Other Server Activities

You need to be aware of the impact of other server activity and maintenance on network activity, especially the following kinds of activity:

- 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.

Login Protocol

A connection can be kept open and shared by various modules within an application, instead of being repeatedly opened and closed.

Single User vs. 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 because of 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 the

long transaction in Figure 20-3, before digging more deeply into the database system to solve an abnormal response time problem.

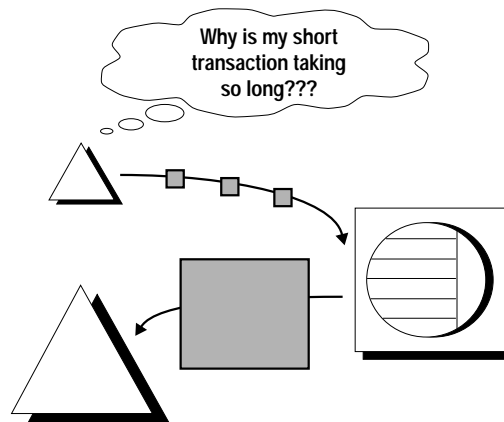


Figure 20-3: Effects of long transactions on other users

Guidelines for Improving Network Performance

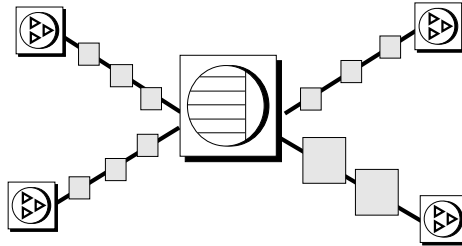
Choose the Right Packet Size for the Task

If most applications send and receive small amounts of data, with a few applications performing larger transfers, here are some guidelines:

- Keep default network packet size small, unless all applications transfer large amounts of data, as shown in Figure 20-4.

- Configure max network packet size and additional network memory just for the applications that need it.

Most applications transfer small amounts of data, a few applications perform large transfers



All applications transfer large amounts of data

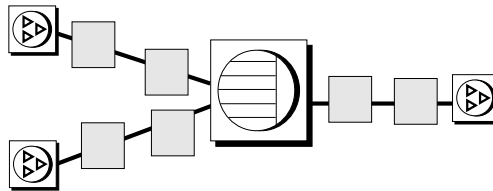


Figure 20-4: Match network packet sizes to application mix

If most of your applications send and receive large amounts of data, increase the default network packet size. This will result in fewer (but larger) transfers.

Isolate Heavy Network Users

Isolate heavy network users from ordinary network users by placing them on a separate network, as shown in Figure 20-5.

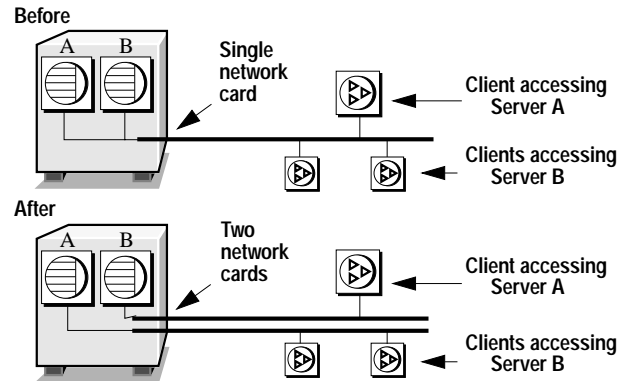


Figure 20-5: Isolating heavy network users

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.

It would be even better to put your Adaptive Servers on different machines.

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 partially full 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 sets the `tcp no delay` configuration parameter to 1.

Configure Multiple Network Listeners

Use two (or more) ports listening for a single Adaptive Server as shown in Figure 20-6. 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.

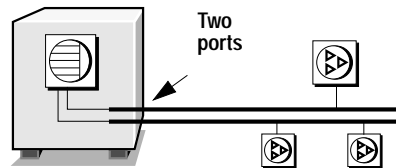


Figure 20-6: Configuring multiple network ports

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

21

How Adaptive Server Uses Engines and CPUs

This chapter explains how Adaptive Server uses 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 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.

This chapter contains the following sections:

- Background Concepts 21-1
- The Single-CPU Process Model 21-4
- The Adaptive Server SMP Process Model 21-10
- Multiple Run Queues 21-12
- A Processing Scenario 21-13
- How the Housekeeper Task Improves CPU Utilization 21-16
- Measuring CPU Usage 21-18
- Enabling Engine-to-CPU Affinity 21-20
- Multiprocessor Application Design Guidelines 21-22

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 over-burdening the operating system. (Multithreaded software can process multiple tasks during the same time period.)

In a system with multiple CPUs, you can enhance performance by configuring Adaptive Server to run using multiple Adaptive Server

instances called **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. Adaptive Server 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. This step and step 6 take place only 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 “Adaptive Server’s Worker Process Model” on page 13-3.
6. Adaptive Server executes the task.
7. If the query was executed in parallel, the task merges the results of the subtasks.
8. 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 queueing, locking, semaphores, and spinlocks to ensure that only one task at a time has access to any common, modifiable data structures. These mechanisms are necessary because Adaptive Server processes multiple queries concurrently. Without these mechanisms, if two or more queries were to access the same data, data integrity would be 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 as operating system processes, because subprocesses use only a small fraction of the operating system resources that a process uses. Multiple processes executing concurrently require more memory and CPU time than multiple subprocesses require. Processes also require operating system resources to switch context (time-share) from one process to the next.

Adaptive Server's 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 do not need any operating system resources after they are launched, and they can share many system resources and structures.

Figure 21-1 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.

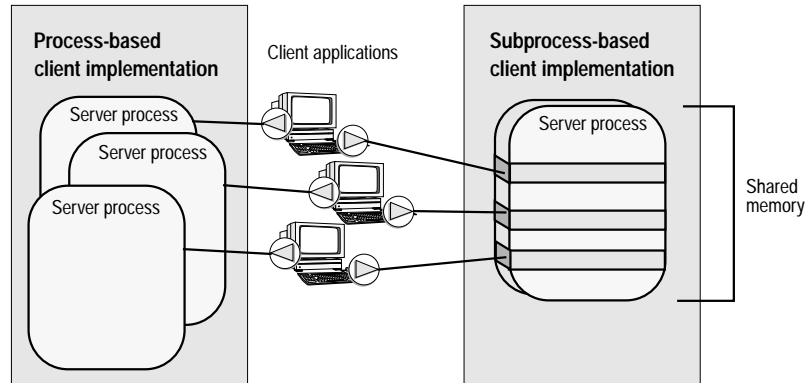


Figure 21-1: Process vs. subprocess architecture

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

The 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 gives an overview of how an Adaptive Server system with a single CPU uses the CPU to process client requests. “The Adaptive Server SMP Process Model” on page 21-10 expands on this discussion to show how an Adaptive Server system with multiple CPUs processes client requests.

Scheduling Engines to the CPU

Figure 21-2 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.

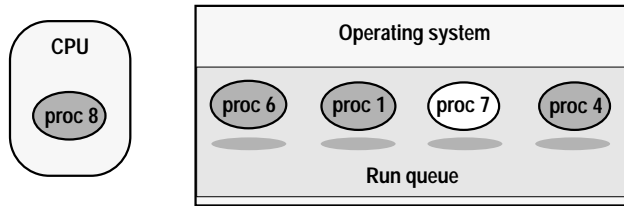


Figure 21-2: Processes queued in the run queue for a single CPU

In a multitasking environment, multiple processes or subprocesses execute concurrently, alternately sharing CPU resources. Figure 21-3 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.

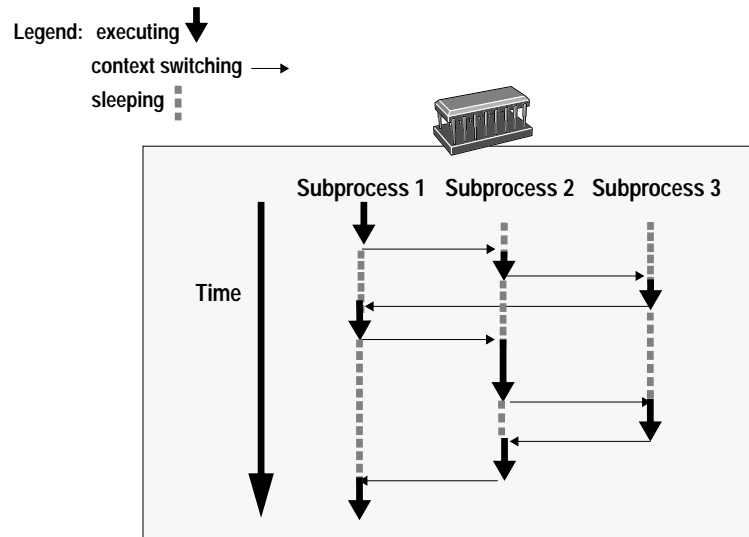


Figure 21-3: Multithreaded processing

Scheduling Tasks to the Engine

Figure 21-4 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 21-2 on page 21-4) to Adaptive Server internal task processing. Adaptive Server, not the operating system, dynamically schedules client tasks from the Adaptive Server 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 an unavailable 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 21-6 discusses how this works in more detail.

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 21-4 also shows the sleep queue with two sleeping tasks, as well as other objects in shared memory (shown in light outline). Tasks are put to sleep while they are waiting for resources or for the results of a disk I/O operation. Adaptive Server uses the task’s stack to keep track of the task’s state from one execution cycle to the next.

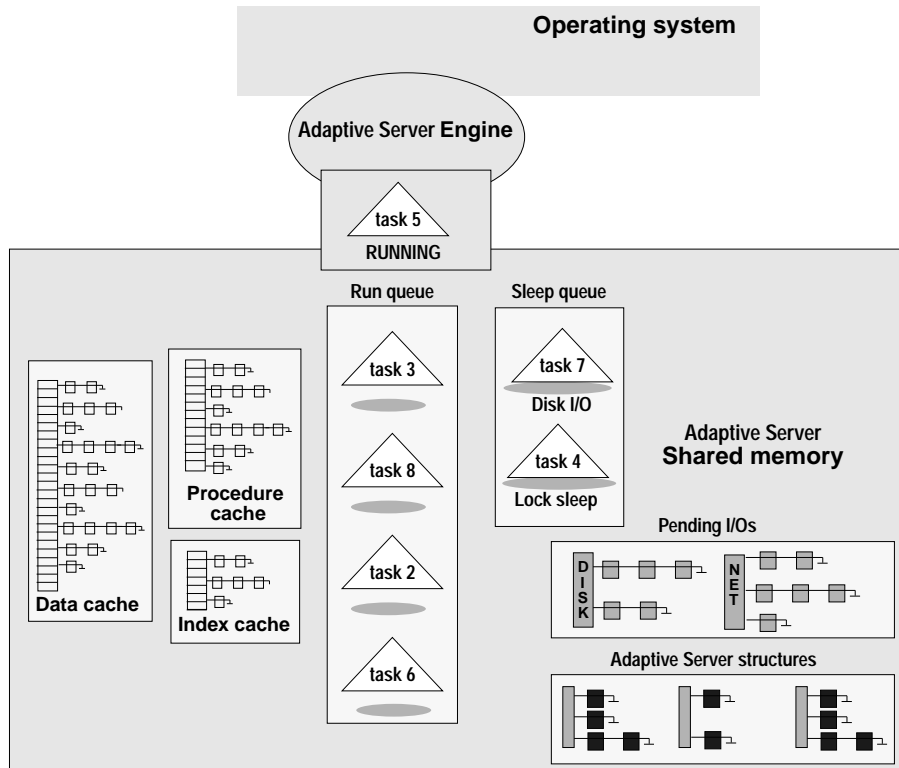


Figure 21-4: Tasks queue up for the Adaptive Server engine

Adaptive Server Execution Task Scheduling

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

Scheduling Client Task Processing Time

The configuration parameter `time slice` keeps tasks that are executing 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, as illustrated in Figure 21-5, using default times for `time slice` (100 clock ticks) and `cpu grace time` (500 clock ticks).

Figure 21-5 shows how Adaptive Server schedules execution time for a task when other tasks are waiting in the run queue. Adaptive Server's scheduler does not force tasks off an Adaptive Server engine. Tasks voluntarily relinquish the engine at a **yield point**, which is a time during which the task does not hold a vital resource such as a spinnlock.

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.

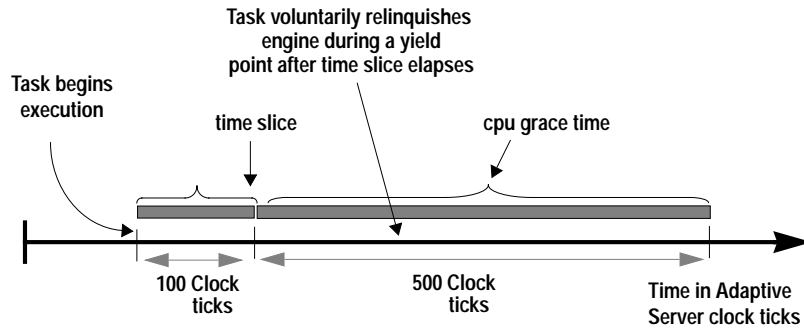


Figure 21-5: Task execution time schedule when other tasks are waiting

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 the end of the time slice interval, Adaptive Server grants the task another processing interval as shown Figure 21-6.

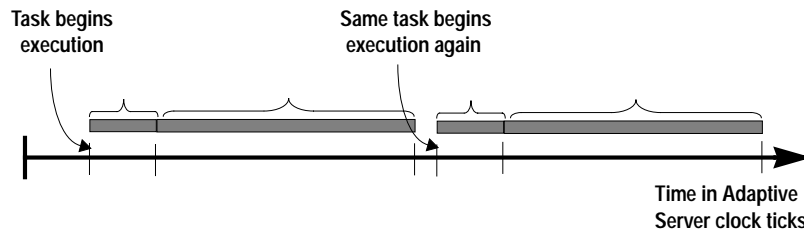


Figure 21-6: Task execution time schedule when no other tasks are waiting

Figure 21-6 shows only two consecutive intervals; however, 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. This causes Adaptive Server to terminate the task with an error.

Figure 21-7 shows a task that fails to relinquish the engine by the end of the `cpu grace time` 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 doubling the value of `cpu grace time`. If the problem persists, call Sybase Technical Support.

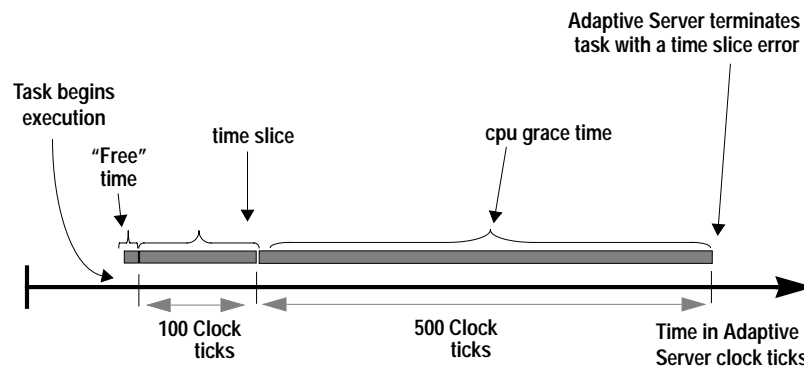


Figure 21-7: A task fails to relinquish the engine within the scheduled time

Guarding 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.

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 the `runnable process search count` parameter affects Adaptive Server's use of CPU cycles, engine yields to the operating system, and blocking network checks. See "Engine

Busy Utilization” on page 24-11 and “Network Checks” on page 24-13.

Tuning Scheduling Parameters

The default value for the `time slice` parameter is 100 clock ticks, 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 could spend too much time switching between tasks, which tends to increase response time. If `time slice` is set too high, CPU-intensive processes might monopolize the CPU, which could increase response time for short tasks. If your applications encounter time slice errors, adjust `cpu grace time`, not `time slice`.

If you decide that you must tune `time slice`, consider the values for the following parameters:

- `cpu grace time` specifies the maximum length of additional time that a user task can execute once its `time slice` has elapsed. If the task does not yield within the grace time, the scheduler terminates it with a time slice error.
- `runnable process search count` specifies the number of times an engine loops, looking for a runnable task, before relinquishing to the CPU.
- `sql server clock tick length` specifies the duration of Adaptive Server’s clock tick in microseconds.
- `i/o polling process count` specifies the number of Adaptive Server tasks that the scheduler allows to execute before checking for disk and/or network I/O completions.

Use `sp_sysmon` to determine how the `time slice` parameter affects voluntary yields by Adaptive Server engines. See “Voluntary Yields” on page 24-23.

The Adaptive Server SMP Process Model

Adaptive Server’s 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 could. SMP is intended for machines with the following features:

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

- Two to 32 processors
- Very high throughput

Scheduling Engines to CPUs

In a system with multiple CPUs, multiple processes can run concurrently. Figure 21-8 represents Adaptive Server engines as the non-shaded 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.

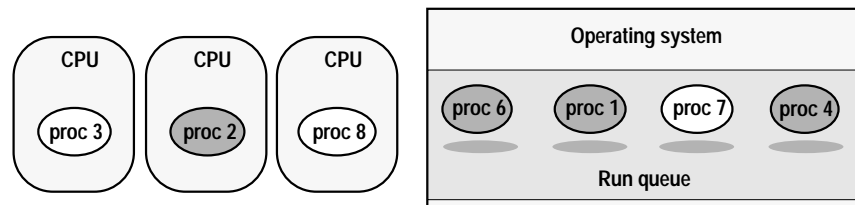


Figure 21-8: Processes queued in the OS run queue for multiple CPUs

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 21-5. The difference is that in the SMP environment, any one of the multiple engines can process the tasks in the run queue.

Figure 21-9 shows tasks (or they could be worker processes) queued up for an Adaptive Server engine. This figure switches from Adaptive Server in the operating system context (as shown in Figure 21-8) to Adaptive Server internal task processing. Figure 21-8 shows

that engines queue up for CPUs, and Figure 21-9 shows that tasks queue up for engines.

The discussion in “Adaptive Server Execution Task Scheduling” on page 21-6 also applies to the SMP environment.

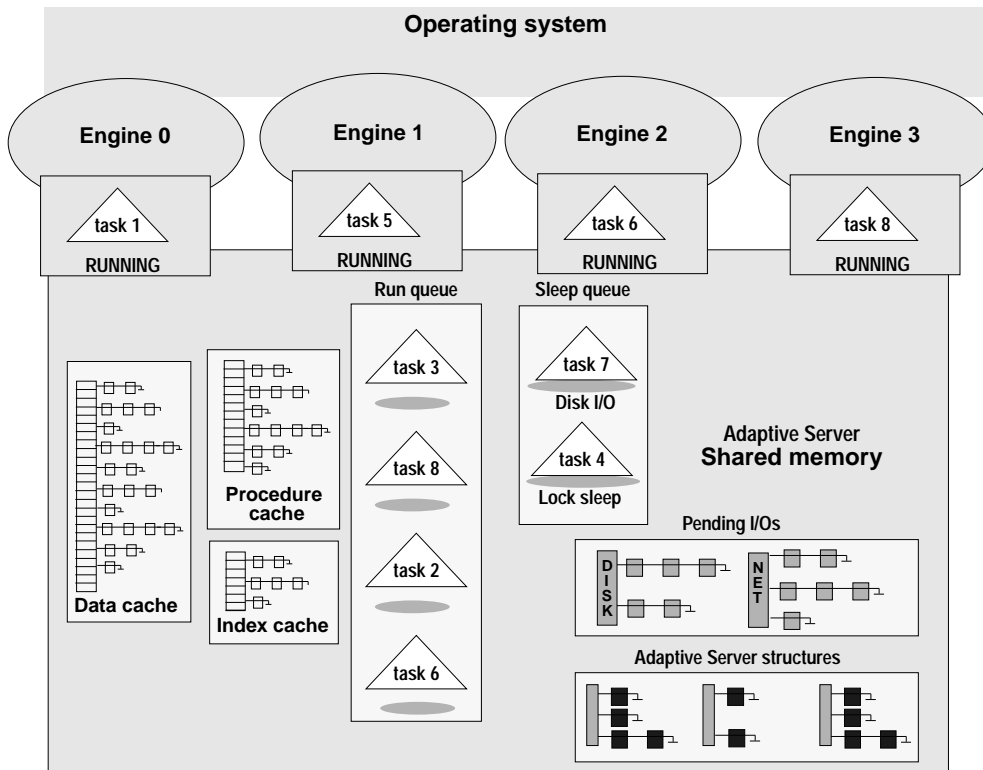


Figure 21-9: Tasks queue up for multiple Adaptive Server engines

Multiple Network Engines

If the SMP system supports network affinity migration, each Adaptive Server engine handles the network I/O for its connections. Engines are numbered sequentially, starting with engine 0. With the exception of engine 0, all engines do the same work. When a user logs into Adaptive Server, engine 0 handles the login to establish packet size, language, character set, and other login settings. After

the login is complete, engine 0 determines which engine is managing the fewest user connections and passes the file handle for the new client task to that engine. That engine then becomes the task's **network engine**.

Because any engine can be a network engine, Adaptive Server processing is fully symmetrical. By allowing each database engine to perform its own network I/O, Adaptive Server scales to the maximum number of network connections that the hardware and operating system can manage, resulting in high throughput. This strategy spreads the network I/O load across multiple hardware network controllers when they are used in large SMP systems. Multiple network engines provide the following additional benefits:

- They increase the number of simultaneous connections beyond 10,000 users without the need for a transaction monitor
- They distribute CPU utilization, increasing overall performance
- They improve query response times by eliminating network bottlenecks

Multiple Run Queues

For most purposes, modeling the run queue as a single component is adequate; however, if there are any execution class assignments, assigned using `sp_bindexclass` and related system procedures, then the run queue component must be modeled as three separate queues with different priorities, as shown in Figure 21-10.

Each run queue has a different priority: low, medium, or high. The queues are used to give preference to higher priority tasks. See Chapter 22, "Distributing Engine Resources Between Tasks," for more information on performance tuning and run queues.

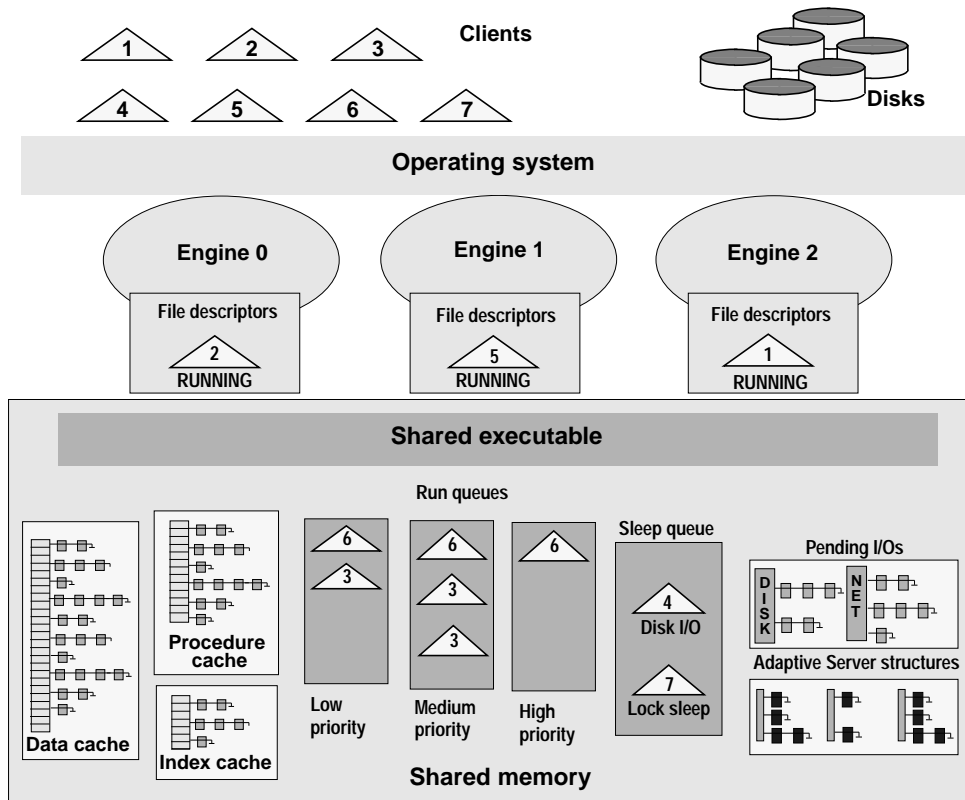


Figure 21-10: Scheduling model when execution classes are assigned

A Processing Scenario

Figure 21-11 shows a conceptual representation of the SMP subsystems, which consist of clients, disks, the operating system, multiple CPUs, and the Adaptive Server executable. (If there were only one engine, rather than three, Figure 21-10 could also represent a single-CPU system.) It also represents the data structures in shared memory:

- Data caches and procedure cache
- Queues for network and disk I/O
- A sleep queue for processes that are waiting for a resource or that are idle

- Run queues for processes that are ready to begin or continue executing

This section describes how an Adaptive Server SMP system manages tasks in a serial execution environment using the subsystems shown in Figure 21-11. The execution cycle for single-processor systems is very similar.

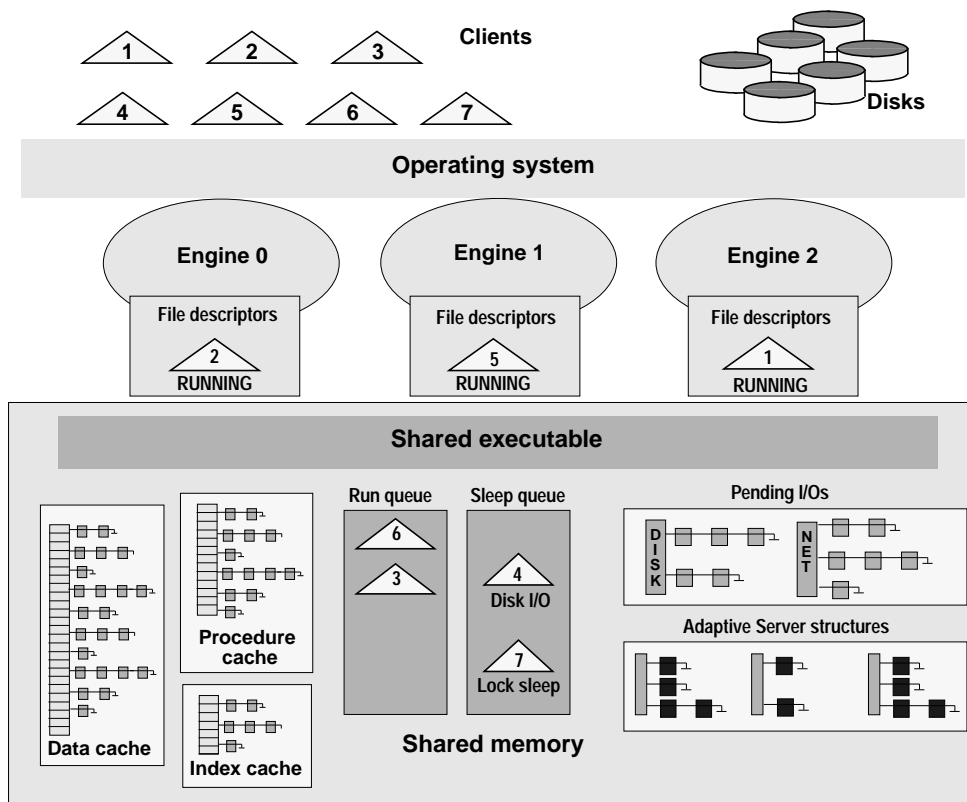


Figure 21-11: Adaptive Server task management in the SMP environment

The difference between the execution cycle for single-processor systems and this SMP example is the network migration, described in step 1 in the following list of processing steps. Also, steps 7 and 8 do not take place in a single-processor system. 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.

Logging In and Assigning a Network Engine

1. When a user logs into Adaptive Server, engine 0 handles the login. It assigns a task structure from the pool of user connections that were allocated at start-up time. It also establishes packet size, language, character set, and other login settings. After the login is complete, engine 0 determines which engine is managing the fewest user connections and passes the file descriptor for the task to that engine. Suppose that Engine 2 has the fewest connections and becomes the network engine for task 5. Task 5 then sleeps while it waits for the client to send a request (not shown in Figure 21-11).

Checking for Client Requests

2. Engine 2 checks for incoming client requests once every clock tick.
3. When Engine 2 finds a command (or query) from the connection for task 5, it wakes up the task and places it on the end of the run queue.

Tasks in the run queue are all in a runnable state. Output from the `sp_who` system procedure lists tasks as “runnable” when the task is in the run queue.

Fulfilling a Client Request

4. When task 5 becomes first in the queue, any idle engine can parse, compile, and begin executing the steps defined in the task’s query plan. Figure 21-11 shows that Engine 1 was the first idle engine and that it is executing the query plan for task 5.

Performing Disk I/O

5. If the task needs to perform disk I/O, it issues an I/O request to the operating system, and Adaptive Server places the task in the sleep queue again.
6. Once each clock tick, the pending disk I/O queue is checked to see whether the task’s I/O has completed. When I/O is complete, the task is moved to the run queue. When the task reaches the head of the run queue, the next available engine resumes execution.

Performing Network I/O

7. When the task needs to return results to the user, it must perform the network write on Engine 2. Engine 1 puts the tasks to sleep on a network write.
8. As soon as the task that Engine 2 is currently executing yields or is put to sleep, Engine 2's scheduler checks to determine whether it has any network tasks pending.
9. Engine 2 issues the network writes, removing the data structures from the network I/O queue.
10. When the write is complete, the task is awakened and placed in the run queue. When the task reaches the head of the queue, it is scheduled on the next available engine.

How the Housekeeper Task Improves CPU Utilization

When Adaptive Server has no user tasks to process, a housekeeper task automatically begins writing dirty buffers to disk. Because these writes are done during the server's idle cycles, they 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).

Side Effects of the Housekeeper Task

If the housekeeper 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 process may improve recovery speed for the database.

In applications that repeatedly update the same database page, the housekeeper task 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 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 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 task at the default parameter setting results in improved performance and recovery speed on most systems.

A dbcc tune option, `deviochar`, controls the size of batches that the housekeeper can write to disk at one time. See “Increasing the Housekeeper Batch Limit” on page 24-85.

Changing the Percentage by Which Writes Can Be Increased

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

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

For example, issue the following command to stop the housekeeper 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 task in order to establish a controlled environment in which only specified user tasks are running. To disable the housekeeper task, set the value of the `housekeeper free write percent` parameter to 0:

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

Allowing the Housekeeper Task to Work Continuously

To allow the housekeeper task to work continuously, 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” section of `sp_sysmon` shows checkpoint information to help you determine the effectiveness of the housekeeper. See “Recovery Management” on page 24-83.

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 1997  5:25PM          Jul 28 1997  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 Reference Manual*.

Using `sp_sysmon` to Measure CPU Usage

The `sp_sysmon` system procedure gives more detailed information than the `sp_monitor` system procedure. 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 Chapter 24, “Monitoring Performance with `sp_sysmon`,” for more information about `sp_sysmon`.

Operating System Commands and CPU Usage

Operating system commands for displaying CPU usage are covered 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.

Multiple CPU Machines

Under Adaptive Server’s SMP (symmetric multiprocessing) architecture, any engine can handle any server task and use any CPU. See “Adaptive Server Task Management for SMP” on page 10-3 of the *System Administration Guide* for a brief description of the process and for information about the system administration tasks and tuning issues for managing SMP configurations.

Determining When to Configure Additional Engines

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

- 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 that corresponds to the following sections in Chapter 24, “Monitoring Performance with `sp_sysmon`.” In particular, study the lines or sections in the output that may reveal points of contention.

For more information, see these sections in Chapter 24, “Monitoring Performance with `sp_sysmon`”:

- “Logical Lock Contention” on page 24-24
- “Address Lock Contention” on page 24-25
- “ULC Semaphore Requests” on page 24-45
- “Log Semaphore Requests” on page 24-46
- “Page Splits” on page 24-51
- “Deadlock Percentage” on page 24-60
- “Spinlock Contention” on page 24-74
- “I/Os Delayed By” on page 24-89

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.

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")
```

results in the following affinity:

Engine	CPU
0	2 (the <i>start_cpu</i> 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 will not be used by Adaptive Server.

To disable CPU affinity, use -1 in place of *start_cpu*, and use *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.

You can check the network I/O load across all Adaptive Server engines using the *sp_sysmon* system procedure. See “Network I/O Management” on page 24-92.

To determine whether your platform supports network affinity migration, consult your operating system documentation or the Adaptive Server installation and configuration guides.

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 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 24-48.

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 24-86 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.

Setting *max_rows_per_page*

The use of *fillfactor* places fewer rows on data and index pages when the index is created, but the *fillfactor* is not maintained. Over time, data modifications can increase the number of rows on a page.

For tables and indexes that experience contention, *max_rows_per_page* provides a permanent way to limit the number of rows on data and index pages.

The `sp_helpindex` system procedure reports the current *max_rows_per_page* setting of indexes. Use the `sp_chgattribute` system procedure to change the *max_rows_per_page* setting.

Setting *max_rows_per_page* to a lower value does not reduce index splitting, and, in most cases, increases the number of index page splits. It can help reduce other lock contention on index pages. If your problem is index page splitting, consider changing the value of the *fillfactor* parameter.

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 19-10 for information on ways to reduce contention.

22

Distributing Engine Resources Between Tasks

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 21, “How Adaptive Server Uses Engines and CPUs.”

This chapter discusses the following topics:

- Using Execution Attributes to Manage Preferred Access to Resources 22-1
- Types of Execution Classes 22-2
- Execution Class Attributes 22-3
- Setting Execution Class Attributes 22-7
- Rules for Determining Precedence and Scope 22-12
- Getting Information About Execution Class Bindings and Attributes 22-12
- Setting Attributes for a Session Only 22-11
- Example Scenario Using Precedence Rules 22-17
- Considerations for Engine Resource Distribution 22-20
- Algorithm for Successfully Distributing Engine Resources 22-22

Using Execution Attributes to Manage Preferred Access to Resources

Most performance-tuning techniques give you control at the system or 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 places tasks in one of three priority run queues. 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 enhancing performance for chosen connections or sessions. In a multiple-application environment, distributing resources can 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 preferable attributes
- *EC2*—has average values of attributes
- *EC3*—has non-preferred values of attributes

Objects associated with execution class *EC2* are given average preference for engine resources. If an execution object is associated with execution class *EC1*, Adaptive Server considers it to be critical and tries to give it preferred access to engine resources. Any execution object associated with execution class *EC3* is considered to be least critical and does not receive engine resources until execution

objects *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 the `sp_bindexclass` system procedure (described in "Assigning Execution Classes" on page 22-7).

User-Defined Execution Classes

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

- *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 22-1. You specify the mix of

attribute values for user-defined execution classes when you create them, using the system procedure `sp_addexclass`.

Table 22-1: Fixed-attribute composition of predefined execution classes

Execution Class (EC) Level	Base Priority Attribute*	Time Slice Attribute (T)**	Engine Affinity Attribute***
<i>EC1</i>	High	Time slice > t	None
<i>EC2</i>	Medium	Time slice = t	None
<i>EC3</i>	Low	Time slice < t	Engine with the highest engine ID number

* See “Base Priority” on page 22-4

** See “Time Slice” on page 22-5

*** See “Task-to-Engine Affinity” on page 22-5

Base Priority

Base priority is the priority assigned by Adaptive Server to an Adaptive Server task when it is created. Base priority refers to the priority of the run queue associated with the task. Assignable values are “high”, “medium”, and “low”. Figure 22-1 shows tasks in the three priority run queues that correspond to the three levels of base priority.

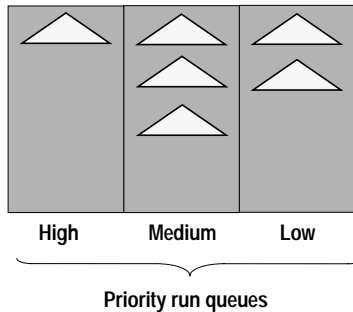


Figure 22-1: Tasks queued in the three priority run queues

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.

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 22-1 on page 22-4, the time slice attribute is different for each predefined execution class. Simply put, *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 slice is based on the value for the time slice configuration parameter, as described in “time slice” on page 11-123 in the *System Administration Guide*. *EC1* execution objects have a time slice value that is greater than the configured value; the time slice of an *EC2* execution object is also greater than the configured value, but its value is less than the time slice value of an *EC1* execution object; and the time slice value of an *EC3* execution object is the same as the configured value.

Currently, Adaptive Server ignores any value you use for the time slice parameter in the `sp_addexclass` system procedure and uses the configured value *n* for all execution classes.

Task-to-Engine Affinity

In a multiprocessor environment running multiple Adaptive Server engines, any available engine can process the next task in the 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.

You can use the engine affinity attribute to override the fixed affinity of a task to an engine associated with predefined execution classes. *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.

While the engine affinity attribute lets you bind a connection or session to a particular Adaptive Server engine, the operating system has control over which CPU runs the Adaptive Server process. In an SMP system, the operating system could switch an Adaptive Server process among CPUs. If your system requires a specific CPU loading, bind the Adaptive Server process (or processes) to a CPU using operating system commands or Adaptive Server's `dbcc tune` command with the `cpuaffinity` parameter. See "Enabling Engine-to-CPU Affinity" on page 21-20 for more information.

► **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 22-2: 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 both predefined and 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_setpsex</code> • <code>sp_clearpsex</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_showpsex</code>

See the *Adaptive Server Reference Manual* for complete descriptions of the system procedures in Table 22-2.

Assigning Execution Classes

The following example illustrates what it means to assign to one execution object (an application in this case) preferred access to resources by associating it with *ECL*.

The example uses the system procedures `sp_addexclass` and `sp_bindexclass`. You set the attributes for user-defined execution classes using the `sp_addexclass` system procedure’s parameters. The syntax is:

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

The syntax for the `sp_bindexclass` system procedure is:

```
sp_bindexclass object_name, object_type,
                scope, class_name
```

Suppose you decide that the “sa” login must get results from `isql` as fast as possible, even if it means a slower response time for a user login using `isql`. You can tell Adaptive Server to give execution preference to login “sa” when it executes `isql` by issuing `sp_bindexclass` with the preferred execution class `EC1`. For example:

```
sp_bindexclass 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 will do what it can to improve response time for the “sa” login with respect to other execution objects running with `EC2` and `EC3` values.

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.

1. Create an engine group using the `sp_addengine` system procedure.

For example, executing `sp_addengine` with the parameters shown in the following statement creates a group called `DS_GROUP` (if there is not already an engine group called `DS_GROUP`). The new group consists 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 the `sp_addexclass` system procedure.

For example, executing `sp_addexclass` with the parameters shown in the following example statement defines a new execution class called `DS` with a priority value of “LOW” and associates it with the engine group `DS_GROUP`. (Adaptive Server currently ignores the third parameter of this system procedure.)

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


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

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

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

In each statement, 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 end result of the preceding example is that the technical support group (not bound to an engine group), which must respond as quickly as possible to customer needs, is given access to more immediate processing resources than the managers, who can afford a little slower response time.

Figure 22-2 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” have no engine affinity and can use all six Adaptive Server engines.

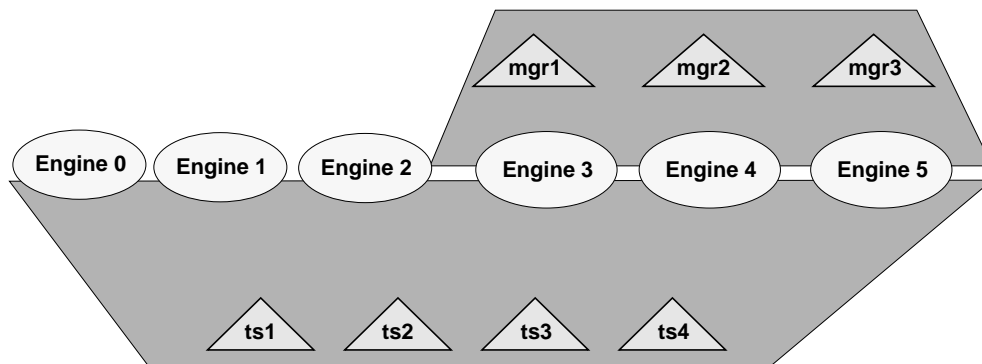


Figure 22-2: An example of engine affinity

How Execution Class Bindings Affect Scheduling

Figure 22-3 shows an example in which four execution objects run tasks. There are two applications—an OLTP application (OLTPAP) and a decision support application (DSAP)—and two logins, “mgr” and “cs3”.

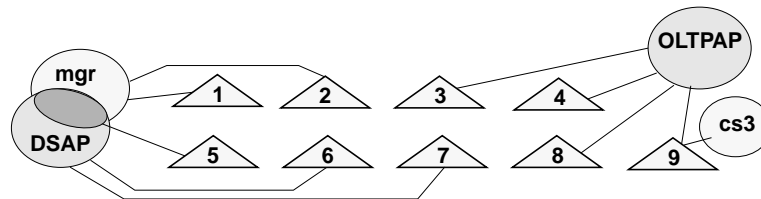


Figure 22-3: Execution objects and their tasks

Execution Class Bindings

The following statement binds OLTPAP with *EC1* attributes, giving higher priority to the tasks running for that application (tasks 3, 4, 8, and 9 in Figure 22-3):

```
sp_bindexeclass OLTPAP, AP, NULL, EC1
```

The overlap of the ovals for the “mgr” login and DSAP results from the following `sp_bindexeclass` statement:

```
sp_bindexeclass mgr, LG, DSAP, EC3
```

This statement stipulates that when the login “mgr” executes DSAP, the task executes with attributes of *EC3*. It will not be able to execute until tasks with *EC1* and *EC2* attributes have finished executing. Task 5 (in Figure 22-3) is running DSAP for the “mgr” login.

Other tasks for the “mgr” login (tasks 1 and 2) and tasks for DSAP (tasks 6 and 7) run with the default attributes of *EC2*.

Effect on Scheduling

Each execution class is associated with a different priority run queue as follows:

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

Adaptive Server executes tasks in the highest priority run queue before it executes those in the medium priority run queue, and it executes tasks in the medium priority run queue before executing those in the low priority run queue.

Figure 22-4 summarizes how the Adaptive Server scheduler queues tasks for the login and applications shown in Figure 22-3.

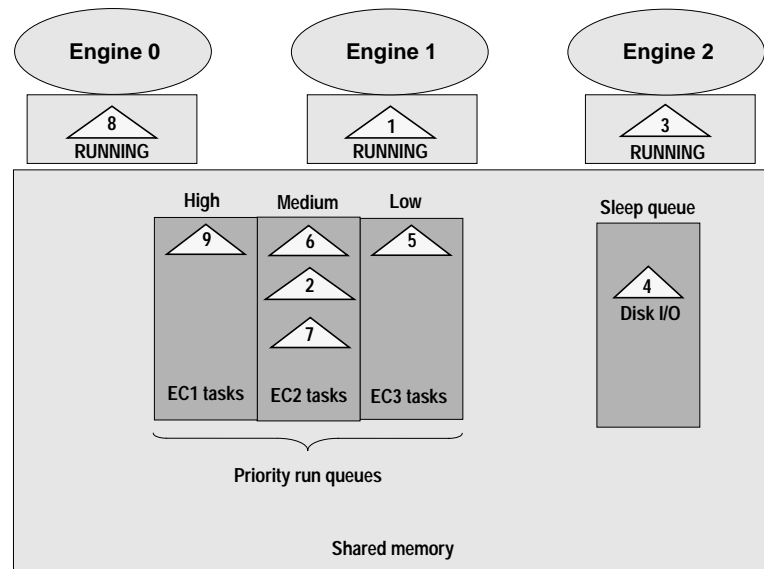


Figure 22-4: How execution class and engine affinity affect execution

What happens if a task has affinity to a particular engine? Assume that rather than having *EC1* attributes, task 9 has a user-defined execution class with high priority and affinity to engine 2. Figure 22-4 still illustrates the scenario, with task 9 first in the high priority run queue. If engine 2 is busy, but engine 1 is not busy, then engine 1 will process task 6, which has no affinity and is first in the medium priority run queue. (Engine 1 would process the next task in the high-priority run queue if there were one.) Although a System Administrator assigned the preferred execution class *EC1* to the OLTP application, engine affinity temporarily lowered task 9's execution precedence to below that of a task with *EC2*.

This effect might be highly undesirable or it might be just what the performance tuner intended. It is possible to assign engine affinity and execution classes in such a way that task priority is not what you intended. It is also possible to make assignments in such a way that

tasks in the lower priority run queues might not ever run—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 only in effect for the duration of the session, whether it ends naturally or is terminated. Setting attributes using `sp_setpsex` does not alter 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.

Getting Information About Execution Class Bindings and Attributes

Adaptive Server stores the information about execution class assignments in the system tables *sysattributes* and *sysprocesses* and supports several system procedures for finding out what assignments have been made. You can use the system procedure `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, the `sp_showcontrolinfo` system procedure displays the complete set of bindings and the composition of all engine groups.

Another system procedure, `sp_showexclass`, displays the attribute values of an execution class or all execution classes. You can also use `sp_showpsex` to see the attributes of all running processes.

Rules for Determining Precedence and Scope

Figuring out 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 could invoke an *EC1* stored procedure. Do both execution objects take *EC3* attributes, do they take *EC1* attributes, or do they take on *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, Different Scopes

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

The 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, as illustrated in Figure 22-5.

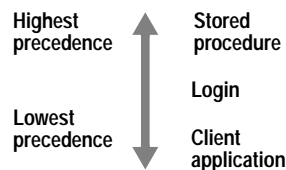


Figure 22-5: Precedence rule

If a stored procedure has a more preferred execution class than that of the client application process invoking it, the client process's precedence 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.

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. Figure 22-6 shows how the precedence rule is applied.

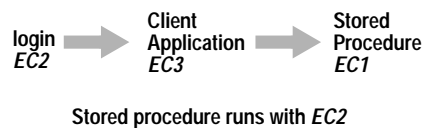


Figure 22-6: Use of the precedence rule

The Scope Rule

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

```
sp_bindexclass 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
```

If you then execute the next statement, any other non-“sa” login or client application that invokes isql will execute with *EC3* attributes:

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

Using the precedence rule for this example, you could say that a client application process servicing 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 the default values shown in the following table:

Entity Type	Attribute Name	Default Value
Client application	Execution class	<i>EC2</i>
Login	Execution class	<i>EC2</i>
Stored procedure	Execution class	<i>EC2</i>

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

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

Examples: Determining Precedence

Each row in Table 22-3 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 22-3: Conflicting attribute values and Adaptive Server assigned values

Execution Class Attributes			Adaptive Server-Assigned Values		
Application (AP) execution class	Login (LG) execution class	Stored procedure (sp_ec) execution class	Application execution class	Login base priority	Stored procedure base priority
<i>EC1</i>	<i>EC2</i>	<i>EC1</i> (<i>EC3</i>)	<i>EC2</i>	Medium	High (Medium)
<i>EC1</i>	<i>EC3</i>	<i>EC1</i> (<i>EC2</i>)	<i>EC3</i>	Low	High (Medium)
<i>EC2</i>	<i>EC1</i>	<i>EC2</i> (<i>EC3</i>)	<i>EC1</i>	High	High (High)
<i>EC2</i>	<i>EC3</i>	<i>EC1</i> (<i>EC2</i>)	<i>EC3</i>	Low	High (Medium)
<i>EC3</i>	<i>EC1</i>	<i>EC2</i> (<i>EC3</i>)	<i>EC1</i>	High	High (High)
<i>EC3</i>	<i>EC2</i>	<i>EC1</i> (<i>EC3</i>)	<i>EC2</i>	Medium	High (Medium)

To test your understanding of the rules of precedence and scope, cover up the “Adaptive Server-Assigned Values” columns in Table 22-3, 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 – a certain client application, AP, is specified as *EC1*
- Column 2 – a particular login, “LG”, is specified as *EC2*
- Column 3 – a stored procedure, sp_ec, is specified as *EC1*

At run time:

- Column 4 – the 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 – the value of column 5 implies a medium base priority for the login.
- Column 6 – the 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 22-7 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.

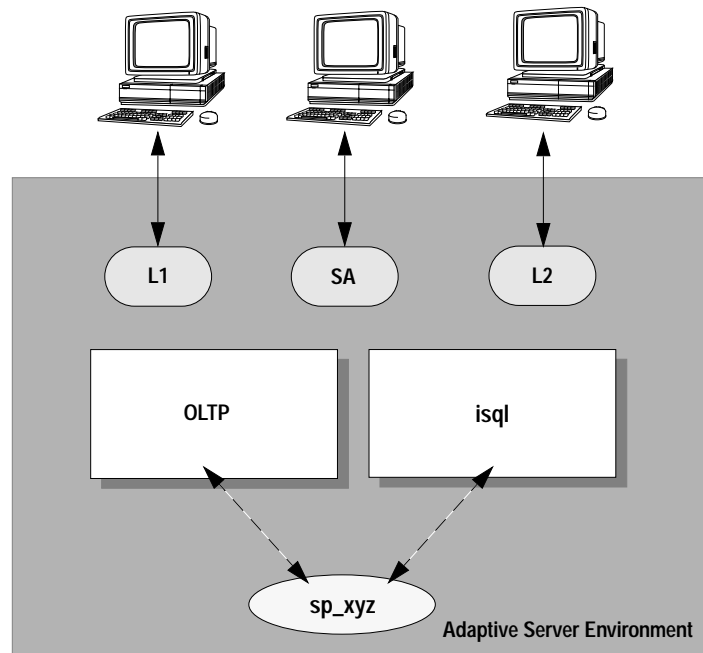


Figure 22-7: Conflict resolution

The rest of this section describes one way to implement the steps discussed in “Algorithm for Successfully Distributing Engine Resources” on page 22-22.

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 22-22 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 22-1 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 22-4: Example analysis of an Adaptive Server environment

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

Configuration

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

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

Execution Characteristics

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

1. A client logs into Adaptive Server as “L1” and invokes 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 into Adaptive Server as “L1” and through *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 into Adaptive Server as “sa” through *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 into Adaptive Server as “L2” and invokes 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 22-5 indicates when assigning an execution precedence might be advantageous for each type of execution object.

Table 22-5: When assigning execution precedence is useful

Execution Object	When Assigning Execution Precedence Is Useful
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. Even if you have as few as two Adaptive Server engines, this is worth trying.

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.

Algorithm for Successfully Distributing Engine Resources

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

The interactions among execution objects in a 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, it is possible to provide guidelines with a progression of steps to use and to discuss the issues commonly related to each step. That is the objective of this section. The steps involved with assigning execution attributes are illustrated in Figure 22-8. A discussion of the steps follows the figure.

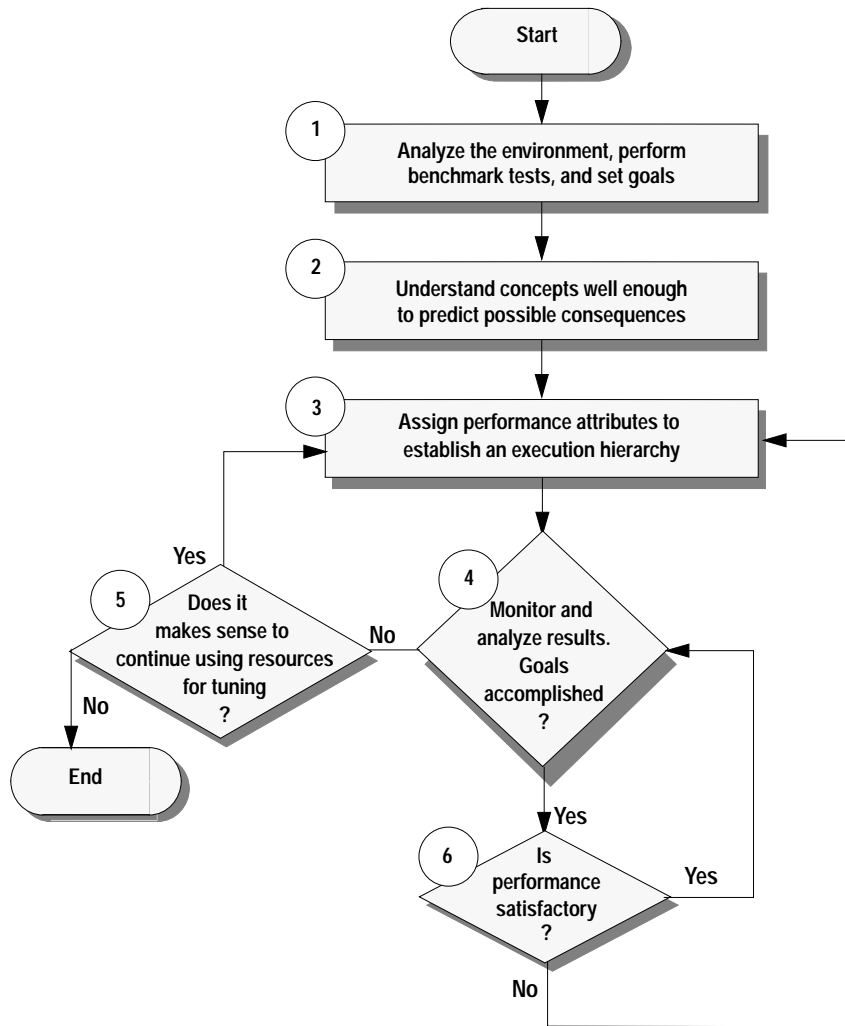


Figure 22-8: Process for assigning execution precedence

Algorithm Guidelines

1. Study the Adaptive Server environment. See “Environment Analysis and Planning” on page 22-24 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 22-3 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. Analyze the running Adaptive Server environment after making execution precedence assignments. See “Results Analysis and Tuning” on page 22-27 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 in Table 22-3 and under “Algorithm for Successfully Distributing Engine Resources” on page 22-22.

Environment analysis and planning involves the following actions:

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

Analyzing the Environment

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 judicious 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 – Analyzing 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 non-critical 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 normally will not use an entire time quantum, and it 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 – Analyzing 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 as follows:

- 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. See Chapter 24, “Monitoring Performance with `sp_sysmon`.” In particular, see “Application Management” on page 24-29.

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. The section “Environment Analysis and Planning” on page 22-24 discusses taking baseline results.
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 24-29.
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 22-24.
Look for inappropriate assumptions and dependencies that you might have missed.
4. Make adjustments to the performance attributes.
5. Fine-tune the results by repeating these steps as many times as necessary.

Monitoring the Environment Over Time

The behavior of an Adaptive Server environment usually varies as the workflow changes during a 24 hour period, over weeks, and over months. Therefore, it is important to monitor the environment to ensure that the system continues to perform well.

23 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.

This chapter contains the following sections:

- Creating or Altering a Database 23-1
- Creating Indexes 23-3
- Backup and Recovery 23-5
- Bulk Copy 23-7
- Database Consistency Checker 23-9
- Using `dbcc tune` (cleanup) 23-10

The common-sense approach is to perform maintenance tasks, whenever possible, at times when your Adaptive Server usage is low. This chapter will help you determine what kind of performance impacts these maintenance activities have on applications and on overall Adaptive Server performance.

Creating or Altering a Database

Creating or altering a database is I/O intensive, and 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, the first page of each 256-page allocation unit and clears database pages.

The following procedures can help 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 configuration parameter `number of large i/o buffers`. The default value for this parameter is 6, allowing parallel I/O to six devices at once. A single `create database` and `alter database` can use up to 8 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 six devices, the first six writes are immediately started in parallel. 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 hugeadb
  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 six 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'  
01:00000:00013:1997/07/26 15:36:17.54 server No disk i/o buffers are  
available for this operation. The total number of buffers is  
controlled by the configuration parameter 'number of large i/o buffers'.  
CREATE DATABASE: allocating 51200 pages on disk 'dev7'  
CREATE DATABASE: allocating 51200 pages on disk 'dev8'  
CREATE DATABASE: allocating 51200 pages on disk 'logdev1'  
CREATE DATABASE: allocating 51200 pages on disk 'logdev2'
```

► **Note**

When create database copies *model*, it uses 2K I/O.

See “number of large i/o buffers” on page 11-21 of the *System Administration Guide* for more information.

Creating 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

The configuration parameters *number of sort buffers* 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. “Configuring Resources for Parallel Sorting” on page 15-6 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. If you need to recover a database, and you have not dumped it 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 the `sorted_data` option requires rearranging the data rows and creating a complete new set of data pages. In some cases, depending on whether the table is partitioned and on what other clauses are used in the `create index` statement, Adaptive Server can skip sorting and/or copying the table's data pages.

When creating an index on a nonpartitioned table, `sorted_data` and the use of any of the following clauses requires copying 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 23-1: Using the `sorted_data` option 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.
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 table is scanned once by following the next page pointers on the data pages. The order of the index rows is checked, and the index tree is built during this single table scan.

If the data pages 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 pages, the space available must be approximately 120 percent of the table size to copy the data and store the index pages.

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, just 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

Backups can be done while a database is active. Clearly, such processing affects other transactions, but do not be afraid to back up critical databases as often as necessary to satisfy the reliability requirements of the system.

See Chapter 20, “Developing a Backup and Recovery Plan,” in 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, the time required to reboot Adaptive Server, 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 16-37.

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 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 copy into a table on Adaptive Server runs fastest when there are no indexes or 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 update indexes for each data insert and to log the changes to the index pages.

To use fast bulk copy, 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 17-30 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:

- The `select into/bulkcopy` option does not have to be set.
- 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.
- Find the optimal network packet size. See “Client Commands for Larger Packet Sizes” on page 20-5.

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 17-30 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, leaving the clustered index in place may be necessary 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 17-25.

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 “Comparing the Performance of **dbcc** Commands” on page 18-17 of the *System Administration Guide* for information about **dbcc** and locking. Also see “Scheduling Database Maintenance at Your Site” on page 18-19 for more information about how to minimize the effects of **dbcc** on user applications.

Using *dbcc tune (cleanup)*

Adaptive Server performs redundant memory clean-up 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 clean-up error check. To turn off error checking, enter:

```
dbcc tune(cleanup,1)
```

This 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)
```


24 Monitoring Performance with *sp_sysmon*

This chapter describes output from *sp_sysmon*, a system procedure that produces Adaptive Server performance data.

This chapter contains the following sections:

- Using *sp_sysmon* 24-2
- Invoking *sp_sysmon* 24-4
- How to Use *sp_sysmon* Reports 24-7
- Sample Interval and Time Reporting 24-10
- Kernel Utilization 24-10
- Worker Process Management 24-16
- Parallel Query Management 24-19
- Task Management 24-21
- Application Management 24-29
- ESP Management 24-35
- Monitor Access to Executing SQL 24-35
- Transaction Profile 24-36
- Transaction Management 24-42
- Index Management 24-48
- Metadata Cache Management 24-55
- Lock Management 24-58
- Data Cache Management 24-65
- Procedure Cache Management 24-81
- Memory Management 24-83
- Recovery Management 24-83
- Disk I/O Management 24-86
- Network I/O Management 24-92

This chapter explains *sp_sysmon* output and gives suggestions for interpreting its output and deducing possible implications. *sp_sysmon* output is most valuable when you have a good understanding of your unique Adaptive Server environment and its specific mix of applications. Otherwise, you may find that *sp_sysmon* output has little relevance.

Using *sp_sysmon*

When you invoke `sp_sysmon`, it clears all accumulated data from internal counters. During the sample period, various Adaptive Server processes increment a set of internal counters. At the end of the sample interval, the procedure reads the counters, prints the report, and stops executing.

The flow diagram below shows the algorithm.

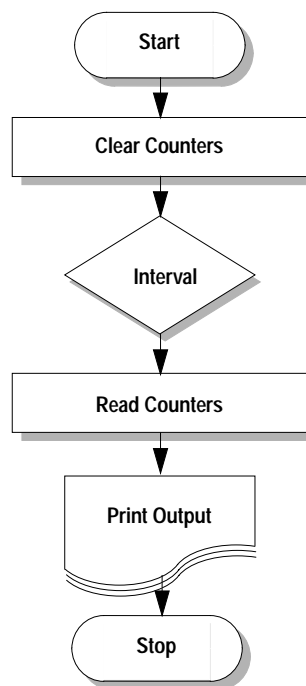


Figure 24-1: `sp_sysmon` execution algorithm

`sp_sysmon` contributes 5 to 7 percent overhead while it runs on a single CPU server, and more on multiprocessor servers. The amount of overhead increases with the number of CPUs.

◆ WARNING!

sp_sysmon and Adaptive Server Monitor use the same internal counters. sp_sysmon resets these counters to 0, producing erroneous output for Adaptive Server Monitor when it is used simultaneously with sp_sysmon.

Also, starting a second execution of sp_sysmon while an earlier execution is running clears all the counters, so the first iteration of reports will be inaccurate.

► Note

sp_sysmon will not produce accurate results on pre-11.0 SQL Servers because many of the internal counters used by sp_sysmon were added in SQL Server release 11.0. In addition, the uses and meanings of many pre-existing counters changed in release 11.0.

When to Run *sp_sysmon*

You can run *sp_sysmon* both before and after tuning Adaptive Server configuration parameters to gather data for comparison. This data gives you a basis for performance tuning and lets you observe the results of configuration changes.

Use *sp_sysmon* when the system exhibits the behavior you want to investigate. For example, if you want to find out how the system behaves under typically loaded conditions, run *sp_sysmon* when conditions are normal and typically loaded. In this case, it would not make sense to run *sp_sysmon* for 10 minutes starting at 7:00 p.m., before the batch jobs begin and after most of the day's OLTP users have left the site. Instead, it would be best to run *sp_sysmon* both during the normal OLTP load and during batch jobs.

In many tests, it is best to start the applications, and then start *sp_sysmon* when the caches have had a chance to reach a steady state. If you are trying to measure capacity, be sure that the amount of work you give the server keeps it busy for the duration of the test. Many of the statistics, especially those that measure data per second, can look extremely low if the server is idle during part of the sample interval.

In general, *sp_sysmon* produces valuable information when you use it:

- Before and after cache or pool configuration changes

- Before and after certain `sp_configure` changes
- Before and after the addition of new queries to your application mix
- Before and after an increase or decrease in the number of Adaptive Server engines
- When adding new disk devices and assigning objects to them
- During peak periods, to look for contention
- During stress tests to evaluate a Adaptive Server configuration for a maximum expected application load
- When performance seems slow or behaves abnormally

It can also help with micro-level understanding of certain queries or applications during development. Some examples are:

- Working with indexes and updates to see if certain updates reported as `deferred_varcol` are resulting direct vs. deferred updates
- Checking caching behavior of particular queries or mix of queries
- Tuning the parameters and cache configuration for parallel index creation

Invoking `sp_sysmon`

There are two ways to use `sp_sysmon`:

- Using a fixed time interval to provide a sample for a specified number of minutes
- Using the `begin_sample` and `end_sample` parameters to start and stop sampling

You can also tailor the output to provide the information you need:

- You can print the entire report.
- You can print just one section of the report, such as “Cache Management” or “Lock Management”.
- You can include application-level detailed reporting for named applications (such as `isql`, `bcp`, or any named application) and for combinations of named applications and user names. (The default is to omit this section.)

Running sp_sysmon for a Fixed Time Interval

To invoke `sp_sysmon`, execute the following command using `isql`:

```
sp_sysmon interval [, section [, applmon]]
```

interval must be in the form “hh:mm:ss”. To run `sp_sysmon` for 10 minutes, use this command:

```
sp_sysmon "00:10:00"
```

The following command prints only the “Data Cache Management” section of the report:

```
sp_sysmon "00:10:00", dcache
```

For information on the *applmon* parameter, see “Specifying the Application Detail Parameter” on page 24-6.

Running sp_sysmon Using begin_sample and end_sample

With the *begin_sample* and *end_sample* parameters, you can invoke `sp_sysmon` to start sampling, issue queries, and end the sample and print the results at any point in time. For example:

```
sp_sysmon begin_sample
execute proc1
execute proc2
select sum(total_sales) from titles
sp_sysmon end_sample
```

► **Note**

During the sample interval, results are stored in signed integer values. Especially on systems with many CPUs and high activity, these counters can overflow. If you see negative results in your `sp_sysmon` output, reduce your sample time.

Specifying Report Sections for *sp_sysmon* Output

To print only a single section of the report, use one of the values listed in Table 24-1 for the second parameter.

Table 24-1: *sp_sysmon* report sections

Report Section	Parameter
Application Management	appmgmt
Data Cache Management	dcache
Disk I/O Management	diskio
ESP Management	esp
Index Management	indexmgmt
Kernel Utilization	kernel
Lock Management	locks
Memory Management	memory
Metadata Cache Management	mdcache
Monitor Access to Executing SQL	monaccess
Network I/O Management	netio
Parallel Query Management	parallel
Procedure Cache Management	pcache
Recovery Management	recovery
Task Management	taskmgmt
Transaction Management	xactmgmt
Transaction Profile	xactsum
Worker Process Management	wpm

Specifying the Application Detail Parameter

The `applmon` parameter to `sp_sysmon` is valid only when you print the entire report or when you request the “Application Management” section by specifying `appmgmt` as the section. It is ignored if you specify it, but request any other section of the report.

The *applmon* parameter must be one of the following:

Parameter	Information Reported
appl_only	CPU, I/O, priority changes and resource limit violations by application name.
appl_and_login	CPU, I/O, priority changes and resource limit violations by application name and login name.
no_appl	Skips the by application or by login section of the report. This is the default.

This example runs *sp_sysmon* for 5 minutes and prints the “Application Management” section, including the application and login detail report:

```
sp_sysmon "00:05:00", appmgmt, appl_and_login
```

Redirecting *sp_sysmon* Output to a File

A full *sp_sysmon* report contains hundreds of lines of output. Use *isql* input and output redirect flags to save the output to a file. See the *Utility Programs* manual for more information on *isql*.

How to Use *sp_sysmon* Reports

sp_sysmon can give you information about Adaptive Server system behavior both before and after tuning. It is important to study the entire report to understand the full impact of the changes you make.

There are several reasons for this. Sometimes removing one performance bottleneck reveals another (see Figure 24-2). It is also possible that your tuning efforts might improve performance in one

area, while actually causing performance degradation in another area.

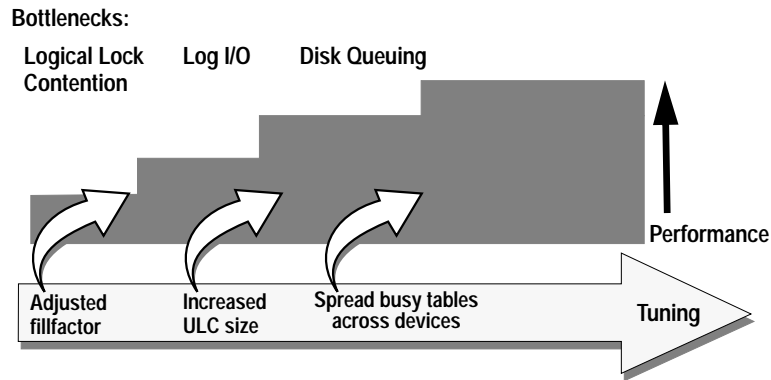


Figure 24-2: Eliminating one bottleneck reveals another

In addition to pointing out areas for tuning work, `sp_sysmon` output is valuable for determining when further tuning will not pay off in additional performance gains. It is just as important to know when to stop tuning Adaptive Server, or when the problem resides elsewhere, as it is to know what to tune.

Other information can contribute to interpreting `sp_sysmon` output:

- Information on the configuration parameters in use, from `sp_configure` or the configuration file
- Information on the cache configuration and cache bindings, from `sp_cacheconfig` and `sp_helpcache`
- Information on disk devices, segments, and the objects stored on them

Reading `sp_sysmon` Output

`sp_sysmon` displays performance statistics in a consistent tabular format. For example, in an SMP environment running nine Adaptive Server engines, the output typically looks like this:


```

Engine Busy Utilization:
Engine 0          98.8 %
Engine 1          98.8 %
Engine 2          97.4 %
Engine 3          99.5 %
Engine 4          98.7 %
Engine 5          98.7 %
Engine 6          99.3 %
Engine 7          98.3 %
Engine 8          97.7 %
-----
Summary:          Total:  887.2 %      Average:  98.6 %

```

Rows

Most rows represent a specific type of activity or event, such as acquiring a lock or executing a stored procedure. When the data is related to CPUs, the rows show performance information for each Adaptive Server engine in the SMP environment. The output above shows nine Adaptive Server engines. Often, when there are groups of related rows, the last row is a summary of totals and an average.

The `sp_sysmon` report indents some rows to show that one category is a subcategory of another. In the following example, “Found in Wash” is a subcategory of “Cache Hits”, which is a subcategory of “Cache Searches”:

```

Cache Searches
Cache Hits          202.1      3.0    12123    100.0 %
  Found in Wash      0.0      0.0      0      0.0 %
Cache Misses        0.0      0.0      0      0.0 %
-----
Total Cache Searches  202.1      3.0    12123

```

Many rows are not printed when the “count” value is 0.

Columns

Unless otherwise stated, the columns represent the following performance statistics:

- “per sec” – average per second during sampling interval
- “per xact” – average per committed transaction during sampling interval
- “count” – total number during the sample interval
- “% of total” – varies, depending on context, as explained for each occurrence

Interpreting *sp_sysmon* Data

When tuning Adaptive Server, the fundamental measures of success appear as increases in throughput and reductions in application response time. Unfortunately, tuning Adaptive Server cannot be reduced to printing these two values. In most cases, your tuning efforts must take an iterative approach, involving a comprehensive overview of Adaptive Server activity, careful tuning and analysis of queries and applications, and monitoring locking and access on an object-by-object basis.

sp_sysmon is a tool that provides a comprehensive overview of system performance. Use Adaptive Server Monitor to pinpoint contention on a per-object basis.

Per Second and Per Transaction Data

Weigh the importance of the per second and per transaction data on the environment and the category you are measuring. The per transaction data is generally more meaningful in benchmarks or in test environments where the workload is well defined.

It is likely that you will find per transaction data more meaningful for comparing test data than per second data alone because in a benchmark test environment, there is usually a well-defined number of transactions, making comparison straightforward. Per transaction data is also useful for determining the validity of percentage results.

Percent of Total and Count Data

The meaning of the “% of total” data varies, depending on the context of the event and the totals for the category. When interpreting percentages, keep in mind that they are often useful for understanding general trends, but they can be misleading when taken in isolation. For example, 50 percent of 200 events is much more meaningful than 50 percent of 2 events.

The “count” data is the total number of events that occurred during the sample interval. You can use count data to determine the validity of percentage results.

Per Engine Data

In most cases, per engine data for a category shows a fairly even balance of activity across all engines. Two exceptions are:

- If you have fewer processes than CPUs, some of the engines will show no activity.
- If most processes are doing fairly uniform activity, such as simple inserts and short selects, and one process performs some I/O intensive operation such as a large bulk copy, you will see unbalanced network and disk I/O.

Total or Summary Data

Summary rows provide an overview of Adaptive Server engine activity by reporting totals and averages.

Be careful when interpreting averages because they can give false impressions of true results when the data is skewed. For example, if one Adaptive Server engine is working 98 percent of the time and another is working 2 percent of the time, a 50 percent average can be misleading.

Sample Interval and Time Reporting

The heading of an `sp_sysmon` report includes the software version, server name, date, the time the sample interval started, the time it completed, and the duration of the sample interval.

```

=====
      Sybase Adaptive Server Enterprise System Performance Report
=====
Server Version: Adaptive Server Enterprise/11.5/P/Sun_svr4/OS 5.5/1/OPT/
Server Name:      tinman
Run Date         Sep 20, 1997
Statistics Cleared at 16:05:40
Statistics Sampled at 16:15:40
Sample Interval   00:10:00

```

Kernel Utilization

“Kernel Utilization” reports Adaptive Server activities. It tells you how busy Adaptive Server engines were during the time that the CPU was available to Adaptive Server, how often the CPU yielded to the operating system, the number of times that the engines checked for network and disk I/O, and the average number of I/Os they found waiting at each check.

Sample Output for Kernel Utilization

The following sample shows `sp_sysmon` output for “Kernel Utilization” in an environment with eight Adaptive Server engines.

Kernel Utilization

 Engine Busy Utilization:

Engine 0	98.5 %
Engine 1	99.3 %
Engine 2	98.3 %
Engine 3	97.2 %
Engine 4	97.8 %
Engine 5	99.3 %
Engine 6	98.8 %
Engine 7	99.7 %

 Summary: Total: 789.0 % Average: 98.6 %

CPU Yields by Engine	per sec	per xact	count	% of total
-----	-----	-----	-----	-----
	0.0	0.0	0	n/a
Network Checks				
Non-Blocking	79893.3	1186.1	4793037	100.0 %
Blocking	1.1	0.0	67	0.0 %
-----	-----	-----	-----	-----
Total Network I/O Checks	79894.4	1186.1	4793104	
Avg Net I/Os per Check	n/a	n/a	0.00169	n/a

Disk I/O Checks

Total Disk I/O Checks	94330.3	1400.4	5659159	n/a
Checks Returning I/O	92881.0	1378.9	5572210	98.5 %
Avg Disk I/Os Returned	n/a	n/a	0.00199	n/a

In this example, the CPU did not yield to the operating system, so there are no detail rows.

Engine Busy Utilization

“Engine Busy Utilization” reports the percentage of time the Adaptive Server Kernel is busy executing tasks on each Adaptive Server engine (rather than time spent idle). The summary row gives the total and the average active time for all engines combined.

The values reported here may differ from the CPU usage values reported by operating system tools. When Adaptive Server has no tasks to process, it enters a loop that regularly checks for network I/O, completed disk I/Os, and tasks in the run queue. Operating

system commands to check CPU activity may show high usage for a Adaptive Server engine because they are measuring the looping activity, while “Engine Busy Utilization” does not include time spent looping—it is considered idle time.

One measurement that cannot be made from inside Adaptive Server is the percentage of time that Adaptive Server had control of the CPU vs. the time the CPU was in use by the operating system. Check your operating system documentation for the correct commands.

See “Engine Busy Utilization” on page 24-11 for an explanation of why operating system commands report different utilization information utilization than Adaptive Server does.

If you want to reduce the time that Adaptive Server spends checking for I/O while idle, you can lower the `sp_configure` parameter `runnable process search count`. This parameter specifies the number of times a Adaptive Server engine loops looking for a runnable task before yielding the CPU. For more information, see “runnable process search count” on page 11-123 of the *System Administration Guide*.

“Engine Busy Utilization” measures how busy Adaptive Server engines were during the CPU time they were given. If the engine is available to Adaptive Server for 80 percent of a 10-minute sample interval, and “Engine Busy Utilization” was 90 percent, it means that Adaptive Server was busy for 7 minutes and 12 seconds and was idle for 48 seconds, as shown in Figure 24-3.

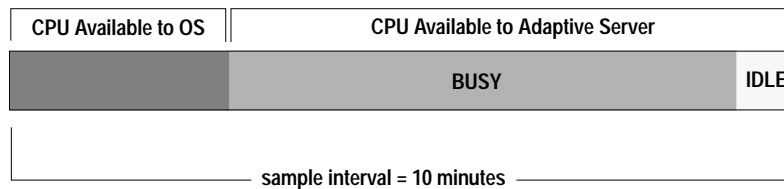


Figure 24-3: How Adaptive Server spends its available CPU time

This category can help you decide whether there are too many or too few Adaptive Server engines. Adaptive Server’s high scalability is due to tunable mechanisms that avoid resource contention. By checking `sp_sysmon` output for problems and tuning to alleviate contention, response time can remain high even at “Engine Busy” values in the 80 to 90 percent range. If values are consistently very high (more than 90 percent), it is likely that response time and throughput could benefit from an additional engine.

The “Engine Busy Utilization” values are averages over the sample interval, so very high averages indicate that engines may be 100 percent busy during part of the interval. When engine utilization is extremely high, the housekeeper process writes few or no pages out to disk (since it runs only during idle CPU cycles.) This means that a checkpoint will find many pages that need to be written to disk, and the checkpoint process, a large batch job, or a database dump is likely to send CPU usage to 100 percent for a period of time, causing a perceptible dip in response time.

If the “Engine Busy Utilization” percentages are consistently high, and you want to improve response time and throughput by adding Adaptive Server engines, carefully check for increased resource contention after adding each engine.

CPU Yields by Engine

“CPU Yields by Engine” reports the number of times each Adaptive Server engine yielded to the operating system. “% of total” data is the percentage of times an engine yielded as a percentage of the combined yields for all engines.

“Total CPU Yields” reports the combined data over all engines.

If the “Engine Busy Utilization” data indicates low engine utilization, use “CPU Yields by Engine” to determine whether the “Engine Busy Utilization” data reflects a truly inactive engine or one that is frequently starved out of the CPU by the operating system.

When an engine is not busy, it yields to the CPU after a period of time related to the `runnable process search count` parameter. A high value for “CPU Yields by Engine” indicates that the engine yielded voluntarily.

If you also see that “Engine Busy Utilization” is a low value, then the engine really is inactive, as opposed to being starved out. See “runnable process search count” on page 11-123 of the *System Administration Guide* for more information.

Network Checks

“Network Checks” includes information about blocking and non-blocking network I/O checks, the total number of I/O checks for the interval, and the average number of network I/Os per network check.

Adaptive Server has two ways to check for network I/O: blocking and non-blocking modes.

Non-Blocking

“Non-Blocking” reports the number of times Adaptive Server performed non-blocking network checks. With non-blocking network I/O checks, an engine checks the network for I/O and continues processing, whether or not it found I/O waiting.

Blocking

“Blocking” reports the number of times Adaptive Server performed blocking network checks.

After an engine completes a task, it loops waiting for the network to deliver a runnable task. After a certain number of loops (determined by the `sp_configure` parameter `runnable process search count`), the Adaptive Server engine goes to sleep after a blocking network I/O.

When an engine yields to the operating system because there are no tasks to process, it wakes up once per clock tick to check for incoming network I/O. If there is I/O, the operating system blocks the engine from active processing until the I/O completes.

If an engine has yielded and is doing blocking checks, it might continue to sleep for a period of time after a network packet arrives. This period of time is referred to as the **latency period**.

You can reduce the latency period by increasing the `runnable process search count` parameter so that the Adaptive Server engine loops for longer periods of time. See “runnable process search count” on page 11-123 of the *System Administration Guide* for more information.

Total Network I/O Checks

“Total Network I/O Checks” reports the number of times an engine polls the sockets for incoming and outgoing packets. This category is helpful when you use it with “CPU Yields by Engine”.

When an engine is idle, it loops while checking for network packets. If “Network Checks” is low and “CPU Yields by Engine” is high, the engine could be yielding too often and not checking the network sockets frequently enough. If the system can afford the overhead, it might be acceptable to yield less often.

Average Network I/Os per Check

“Avg Net I/Os per Check” reports the average number of network I/Os (both sends and receives) per check for all Adaptive Server engine checks that took place during the sample interval.

The `sp_configure` parameter `i/o polling process count` specifies the maximum number of processes that Adaptive Server runs before the scheduler checks for disk and/or network I/O completions. Tuning `i/o polling process count` affects both the response time and throughput of Adaptive Server. See “i/o polling process count” on page 11-109 of the *System Administration Guide*.

If Adaptive Server engines check frequently, but retrieve network I/O infrequently, you can try reducing the frequency for network I/O checking.

Disk I/O Checks

This section reports the total number of disk I/O checks, and the number of checks returning I/O.

Total Disk I/O Checks

“Total Disk I/O Checks” reports the number of times an engine checked disk I/O.

When a task needs to perform I/O, the Adaptive Server engine running that task immediately issues an I/O request and puts the task to sleep, waiting for the I/O to complete. The Adaptive Server engine processes other tasks, if any, but also uses a scheduling loop to check for completed I/Os. When the engine finds completed I/Os, it moves the task from the sleep queue to the run queue.

Checks Returning I/O

“Checks Returning I/O” reports the number of times that a requested I/O had completed when an engine checked for disk I/O.

For example, if an engine checks for expected I/O 100,000 times, this average indicates the percentage of time that there actually was I/O pending. If, of those 100,000 checks, I/O was pending 10,000 times, then 10 percent of the checks were effective, and the other 90 percent were overhead. However, you should also check the average number of I/Os returned per check and how busy the engines were during

the sample interval. If the sample includes idle time, or the I/O traffic is “bursty,” it is possible that during a high percentage of the checks were returning I/O during the busy period.

If the results in this category seem low or high, you can configure *i/o polling process count* to increase or decrease the frequency of the checks. See “i/o polling process count” on page 11-109 in the *System Administration Guide*.

Average Disk I/Os Returned

“Avg Disk I/Os Returned” reports the average number of disk I/Os returned over all Adaptive Server engine checks combined.

Increasing the amount of time that Adaptive Server engines wait between checks could result in better throughput because Adaptive Server engines can spend more time processing if they spend less time checking for I/O. However, you should verify this for your environment. Use the *sp_configure* parameter *i/o polling process count* to increase the length of the checking loop. See “i/o polling process count” on page 11-109 in the *System Administration Guide*.

Worker Process Management

“Worker Process Management” reports the use of worker processes, including the number of worker process requests that were granted and denied and the success and failure of memory requests for worker processes. You need to analyze this output in combination with the information reported under “Parallel Query Management” on page 24-19.

Sample Output for Worker Process Management

```

Worker Process Management
-----

```

	per sec	per xact	count	% of total
	-----	-----	-----	-----
Worker Process Requests				
Requests Granted	0.1	8.0	16	100.0 %
Requests Denied	0.0	0.0	0	0.0 %

Total Requests	0.1	8.0	16	
Requests Terminated	0.0	0.0	0	0.0 %
Worker Process Usage				
Total Used	0.4	39.0	78	n/a
Max Ever Used During Sample	0.1	12.0	24	n/a
Memory Requests for Worker Processes				
Succeeded	4.5	401.0	802	100.0 %
Failed	0.0	0.0	0	0.0 %
Avg Mem Ever Used by a WP				
(in bytes) n/a	n/a	311.7	n/a	n/a

Worker Process Requests

This section reports requests for worker processes and worker process memory. A parallel query may make one or two requests for worker processes: the first for accessing data and the second for parallel sort, if chosen by the sort manager. The `create index` command also uses worker processes.

The “Requests Granted” and “Requests Denied” rows show how many requests were granted and how many requests were denied due to a lack of available worker processes at execution time.

To see the number of adjustments made to the number of worker processes, see “Parallel Query Usage” on page 24-20.

“Requests Terminated” reports the number of times a request was terminated by user action, such as pressing Control-c, that cancelled the query.

Worker Process Usage

In this section, “Total Used” reports the total number of worker processes used during the sample interval. “Max Ever Used During

Sample” reports the highest number in use at any time during `sp_sysmon`’s sampling period. You can use “Max Ever Used During Sample” to set the configuration parameter `number of worker processes`.

Memory Requests for Worker Processes

This section reports how many requests were made for memory allocations for worker processes, how many of those requests succeeded and how many failed. Memory for worker processes is allocated from a memory pool configured with the parameter `memory per worker process`. If “Failed” is a nonzero value, you may need to increase the value of `memory per worker process`.

Avg Mem Ever Used by a WP

This row reports the maximum average memory used by all active worker processes at any time during the sample interval. Each worker process requires memory, principally for exchanging coordination messages. This memory is allocated by Adaptive Server from the global memory pool. The size of the pool is determined by multiplying the two configuration parameters, `number of worker processes` and `memory per worker process`. If `number of worker processes` is set to 50, and `memory per worker process` is set to the default value of 1024 bytes, 50K is available in the pool. Increasing `memory per worker process` to 2048 bytes would require 50K of additional memory.

At start-up, certain static structures are created for each worker process. While worker processes are in use, additional memory is allocated from the pool as needed and deallocated when not needed. The average value printed is the average for all of the static and dynamically memory allocated for all worker processes, divided by the number of worker processes actually in use during the sample interval.

If a large number of worker processes are configured, but only a few are in use during the sample interval, the value printed may be inflated, due to averaging in the static memory for unused processes.

If “Avg Mem” is close to the value set by `memory per worker process` and the number of worker processes in “Max Ever Used During Sample” is close to the number configured, you may want to increase the value of the parameter. If a worker process needs memory from the pool, and no memory is available, the process prints an error message and exits.

► Note

For most parallel query processing, the default value of 1024 is more than adequate. The exception is `dbcc checkstorage`, which can use up 1792 bytes if only one worker process is configured. If you are using `dbcc checkstorage`, and `number of worker processes` is set to 1, you may want to increase `memory per worker process`.

Parallel Query Management

“Parallel Query Management” reports the execution of parallel queries. It reports the total number of parallel queries, how many times the number of worker processes was adjusted at run time, and reports on the granting of locks during merges and sorts.

Sample Output for Parallel Query Management

Parallel Query Management

```

-----
Parallel Query Usage      per sec  per xact  count  % of total
-----
Total Parallel Queries    0.1      8.0      16     n/a
WP Adjustments Made
  Due to WP Limit         0.0      0.0      0      0.0 %
  Due to No WPs           0.0      0.0      0      0.0 %

Merge Lock Requests      per sec  per xact  count  % of total
-----
Network Buffer Merge Locks
  Granted with no wait    4.9      438.5    877    56.2 %
  Granted after wait      3.7      334.5    669    42.9 %

Result Buffer Merge Locks
  Granted with no wait    0.0      0.0      0      0.0 %
  Granted after wait      0.0      0.0      0      0.0 %

Work Table Merge Locks
  Granted with no wait    0.1      7.0      14     0.9 %
  Granted after wait      0.0      0.0      0      0.0 %
-----
Total # of Requests       8.7      780.0    1560

Sort Buffer Waits         per sec  per xact  count  % of total
-----
Total # of Waits         0.0      0.0      0      n/a

```

Parallel Query Usage

“Total Parallel Queries” reports the total number of queries eligible to be run in parallel. The optimizer determines the best plan, deciding whether a query should be run serially or in parallel and how many worker processes should be used for parallel queries.

“WP Adjustments Made” reports how many times the number of worker processes recommended by the optimizer had to be adjusted at run time. The two possible causes are reported:

- “Due to WP Limit” indicates the number of requests for which the number of worker processes was reduced. These queries ran in parallel, but with fewer worker processes than recommended by the optimizer.
- “Due to No WPs” indicates how many requests found no worker processes available at run time or found too few worker processes to be useful for a particular query. These queries were run serially.

If either of these values is a nonzero value, and the sample was taken at a time of typical load on your system, you may want to increase the number of worker processes configuration parameter.

Merge Lock Requests

“Merge Lock Requests” reports the number of parallel merge lock requests that were made, how many were granted immediately, and how many had to wait for each type of merge. The three merge types are:

- “Network Buffer Merge Locks” –reports contention for the network buffers that return results to clients.
- “Result Buffer Merge Locks” –reports contention for the result buffers used to process results for ungrouped aggregates and nonsorted, nonaggregate variable assignment results.
- “Work Table Merge Locks” –reports contention for locks while results from work tables were being merge.

“Total # of Requests” prints the total of the three types of merge requests.

Sort Buffer Waits

This section reports contention for the sort buffers used for parallel sorts. Parallel sort buffers are used by:

- Producers – the worker processes returning rows from parallel scans
- Consumers – the worker processes performing the parallel sort

If the number of waits is high, you can configure number of sort buffers to a higher value. See “Sort Buffer Configuration Guidelines” on page 15-11 for guidelines.

Task Management

“Task Management” provides information on opened connections, task context switches by engine, and task context switches by cause.

Sample Output for Task Management

The following sample shows `sp_sysmon` output for the “Task Management” categories.

Task Management	per sec	per xact	count	% of total
Connections Opened	0.0	0.0	0	n/a
Task Context Switches by Engine				
Engine 0	94.8	0.8	5730	10.6 %
Engine 1	94.6	0.8	5719	10.6 %
Engine 2	92.8	0.8	5609	10.4 %
Engine 3	105.0	0.9	6349	11.7 %
Engine 4	101.8	0.8	6152	11.4 %
Engine 5	109.1	0.9	6595	12.2 %
Engine 6	102.6	0.9	6201	11.4 %
Engine 7	99.0	0.8	5987	11.1 %
Engine 8	96.4	0.8	5830	10.8 %
Total Task Switches:	896.1	7.5	54172	
Task Context Switches Due To:				
Voluntary Yields	69.1	0.6	4179	7.7 %
Cache Search Misses	56.7	0.5	3428	6.3 %
System Disk Writes	1.0	0.0	62	0.1 %
I/O Pacing	11.5	0.1	695	1.3 %
Logical Lock Contention	3.7	0.0	224	0.4 %

Address Lock Contention	0.0	0.0	0	0.0 %
Log Semaphore Contention	51.0	0.4	3084	5.7 %
Group Commit Sleeps	82.2	0.7	4971	9.2 %
Last Log Page Writes	69.0	0.6	4172	7.7 %
Modify Conflicts	83.7	0.7	5058	9.3 %
I/O Device Contention	6.4	0.1	388	0.7 %
Network Packet Received	120.0	1.0	7257	13.4 %
Network Packet Sent	120.1	1.0	7259	13.4 %
SYSINDEXES Lookup	0.0	0.0	0	0.0 %
Other Causes	221.6	1.8	13395	24.7 %%

Connections Opened

“Connections Opened” reports the number of connections opened to Adaptive Server. It includes any type of connection, such as client connections and remote procedure calls. It only counts connections that were started during the sample interval. Connections that were established before the interval started are not counted here.

This data is provides a general understanding of the Adaptive Server environment and the work load during the interval. This data can also be useful for understanding application behavior—it can help determine if applications repeatedly open and close connections or perform multiple transactions per connection.

Task Context Switches by Engine

“Task Context Switches by Engine” reports the number of times each Adaptive Server engine switched context from one user task to another. “% of total” reports the percentage of Adaptive Server engine task switches for each Adaptive Server engine as a percentage of the total number of task switches for all Adaptive Server engines combined.

“Total Task Switches” summarizes task-switch activity for all engines on SMP servers. You can use “Total Task Switches” to observe the effect of controlled reconfigurations. You might reconfigure a cache or add memory if tasks appear to block on cache search misses and to be switched out often. Then, check the data to see if tasks tend to be switched out more or less often.

Task Context Switches Due To

“Task Context Switches Due To” reports the number of times that Adaptive Server switched context for a number of common reasons.

“% of total” reports the percentage of times the context switch was due to each specific cause as a percentage of the total number of task context switches for all Adaptive Server engines combined.

“Task Context Switches Due To” provides an overview of the reasons that tasks were switched off engines. The possible performance problems shown in this section can be investigated by checking other `sp_sysmon` output, as indicated below in the sections that describe the causes.

For example, if most of the task switches are caused by physical I/O, try minimizing physical I/O, by adding more memory or reconfiguring caches. However, if lock contention causes most of the task switches, check the locking section of your report. See “Lock Management” on page 24-58 for more information.

Voluntary Yields

“Voluntary Yields” reports the number of times a task completed or yielded after running for the configured amount of time. The Adaptive Server engine switches context from the task that yielded to another task.

The configuration parameter `time slice` sets the amount of time that a process can run. A CPU-intensive task that does not switch out due to other causes yields the CPU at certain “yield points” in the code, in order to allow other processes a turn on the CPU. See “time slice” on page 11-127 of the *System Administration Guide* for more information.

A high number of voluntary yields indicates that there is not much contention. If this is consistently the case, consider increasing the time slice configuration parameter.

Cache Search Misses

“Cache Search Misses” reports the number of times a task was switched out because a needed page was not in cache and had to be read from disk. For data and index pages, the task is switched out while the physical read is performed.

See “Data Cache Management” on page 24-65 for more information about the cache-related parts of the `sp_sysmon` output.

System Disk Writes

“Disk Writes” reports the number of times a task was switched out because it needed to perform a disk write or because it needed to access a page that was being written by another process, such as the housekeeper or the checkpoint process.

Most Adaptive Server writes happen asynchronously, but processes sleep during writes for page splits, recovery, and OAM page writes.

If this value reported by your system seems high, check the value reported for page splits in your report to see if the problem is caused by data pages and index page splits. See “Page Splits” on page 24-51 for more information.

If the high value for system disk writes is not caused by page splitting, you cannot affect this value by tuning.

I/O Pacing

“I/O Pacing” how many times an I/O intensive tasks was switched off then engine due to exceeding an I/O batch limit. Adaptive Server paces the number of disk writes that it issues in order to keep from flooding the disk I/O subsystems during certain operations that need to perform large amounts of I/O. Two examples are the checkpoint process and transaction commits that write a large number of log pages. The task is switched out and sleeps until the batch of writes completes and then wakes up and issues another batch.

By default, the number of writes per batch is set to 10. You may want to increase the number of writes per batch if:

- You have a high-throughput, high-transaction environment with a large data cache
- Your system is not I/O bound

Valid values are from 1 to 50. This command sets the number of writes per batch to 20:

```
dbcc tune (maxwritedes, 20)
```

Logical Lock Contention

“Logical Lock Contention” reports the number of times a task was switched out because of contention over database locks, such as table and page locks.

Investigate lock contention problems by checking the transaction detail and lock management sections of the report. See “Transaction Detail” on page 24-39 and “Lock Management” on page 24-58. Check to see if your queries are doing deferred and direct expensive updates, which can cause additional index locks. See “Updates” on page 24-41.

For additional help on locks and lock contention, check the following sources:

- “Types of Locks in Adaptive Server” on page 5-3 provides information about types of page and table locks.
- “Reducing Lock Contention” on page 5-33 provides pointers on reducing lock contention.
- Chapter 7, “Indexing for Performance,” provides information on indexes and query tuning. In particular, use indexes to ensure that updates and deletes do not lead to table scans and exclusive table locks.

Address Lock Contention

“Address Lock Contention” reports the number of times a task was switched out because of memory address locks. Adaptive Server acquires address locks on index pages, OAM pages and allocation pages, during updates, and sometimes on data pages when page splits are performed. Address lock contention tends to have more implications in a high throughput environment.

Log Semaphore Contention

“Log Semaphore Contention” reports the number of times a task was switched out because it needed to acquire the transaction log semaphore held by another task. This applies to SMP systems only.

High contention for the log semaphore could indicate that the user log cache (ULC) is too small. See “Transaction Management” on page 24-42. If you decide that the ULC is correctly sized, then think about how to minimize the number of log writes by making application changes.

Another area to check is disk queuing on the disk used by the transaction log. See “Disk I/O Management” on page 24-86.

Also see “Engine Busy Utilization” on page 24-11. If engine utilization reports a low value, and response time is within acceptable limits, consider reducing the number of engines. Running

with fewer engines reduces contention by decreasing the number of tasks trying to access the log simultaneously.

Group Commit Sleeps

“Group Commit Sleeps” reports the number of times a task performed a transaction commit and was put to sleep until the log was written to disk. Compare this value to the number of committed transactions. See “Committed Transactions” on page 24-37. If the transaction rate is low, a higher percentage of tasks wait for “Group Commit Sleeps.”

If there are a significant number of tasks resulting in “Group Commit Sleeps,” and the log I/O size is greater than 2K, a smaller log I/O size can help to reduce commit time by causing more frequent page flushes. Flushing the page wakes up tasks sleeping on the group commit.

In high throughput environments, a large log I/O size is very important to prevent problems in disk queuing on the log device. A high percentage of group commit sleeps should not be regarded as a problem.

Other factors that can affect group commit sleeps are the size of the run queue and the speed of the disk device on which the log resides.

When a task commits, its log records are flushed from its user log cache to the current page of the transaction log in cache. If the page (or pages, if a large log I/O size is configured) is not full, the task is switched out and placed on the end of the run queue. The task wakes up when:

- Another process fills the log page(s), and flushes the log
- When the task reaches the head of the run queue, and no other process has flushed the log page

For more information on setting the log I/O size, see “Choosing the I/O Size for the Transaction Log” on page 16-29.

Last Log Page Writes

“Last Log Page Writes” reports the number of times a task was switched out because it was put to sleep while writing the last log page.

The task switched out because it was responsible for writing the last log page, as opposed to sleeping while waiting for some other task to

write the log page, as described in “Group Commit Sleeps” on page 24-26.

If this value is high, review “Avg # Writes per Log Page” on page 24-47 to determine whether Adaptive Server is repeatedly writing the same last page to the log. If the log I/O size is greater than 2K, reducing the log I/O size might reduce the number of unneeded log writes.

Modify Conflicts

“Modify Conflicts” reports the number of times that a task tried to get exclusive access to a page that was held by another task under a special lightweight protection mechanism. For certain operations, Adaptive Server uses these lightweight protection mechanisms to gain exclusive access to a page without using actual page locks. Examples are access to some system tables and dirty reads. These processes need exclusive access to the page, even though they do not modify it.

I/O Device Contention

“I/O Device Contention” reports the number of times a task was put to sleep while waiting for a semaphore for a particular device.

When a task needs to perform physical I/O, Adaptive Server fills out the block I/O structure and links it to a per-engine I/O queue. If two Adaptive Server engines request an I/O structure from the same device at the same time, one of them sleeps while it waits for the semaphore it needs.

If there is significant contention for I/O device semaphores, try reducing it by redistributing the tables across devices or by adding devices and moving tables and indexes to them. See “Spreading Data Across Disks to Avoid I/O Contention” on page 17-5 for more information.

Network Packet Received

When task switching is reported by “Network Packet Received,” the task switch is due to one of these causes:

- A task received part of a multipacket Tabular Data Stream (TDS) batch and was switched out waiting for the client to send the next TDS packet of the batch, or

- A task completely finished processing a command and was put into a receive sleep state while waiting to receive the next command or packet from the client.

If “Network Packet Received” is high, see “Network I/O Management” on page 24-92 for more information about network I/O. Also, you can configure the network packet size for all connections or allow certain connections to log in using larger packet sizes. See “Changing Network Packet Sizes” on page 20-3 and “default network packet size” on page 11-72 in the *System Administration Guide*.

Network Packet Sent

“Network Packet Sent” reports the number of times a task went into a send sleep state while waiting for the network to send each TDS packet to the client.

The TDS model determines that there can be only one outstanding TDS packet per connection at any one point in time. This means that the task sleeps after each packet it sends.

If there is a lot of data to send, and the task is sending many small packets (512 bytes per packet), the task could end up sleeping a number of times. The TDS data packet size is configurable, and different clients can request different packet sizes. For more information, see “Changing Network Packet Sizes” on page 20-3 and “default network packet size” on page 11-72 in the *System Administration Guide*.

If “Network Packet Sent” is a major cause of task switching, see “Network I/O Management” on page 24-92 for more information.

SYSINDEXES Lookup

“SYSINDEXES Lookup” reports the number of times a task went to sleep while waiting for another task to release control of a page in the *sysindexes* table. This data is meaningful in SMP environments only.

Other Causes

“Other Causes” reports the number of tasks switched out for any reasons not described above. In a well-tuned server, this value may rise as tunable sources of task switching are reduced.

Application Management

“Application Management” reports execution statistics for user tasks. Adaptive Server schedules execution by placing user tasks on three internal run queues. Each run queue has a different execution priority with respect to the others.

This section is especially useful if you plan to tune applications by setting execution attributes and assigning engine affinity. Before making any adjustments to applications, logins, or stored procedures, run `sp_sysmon` during periods of typical load, and familiarize yourself with the statistics in this section. For related background information, see Chapter 22, “Distributing Engine Resources Between Tasks.”

Requesting Detailed Application Information

If you request information about specific tasks using the `sp_sysmon` parameter, `applmon`, `sp_sysmon` output gives statistics specific to each application individually in addition to summary information. You can choose to display detailed application information in one of two ways:

- Application and login information (using the `sp_sysmon` parameter `appl_and_login`)— `sp_sysmon` prints a separate section for each login and the applications it is executing.
- Application information only (using the `sp_sysmon` parameter, `appl_only`)— `sp_sysmon` prints a section for each application, which combines data for all of the logins that are executing it.

For example, if 10 users are logged in with `isql`, and 5 users are logged in with an application called `sales_reports`, requesting “application and login” information prints 15 detail sections. Requesting “application only” information prints 2 detail sections, one summarizing the activity of all `isql` users, and the other summarizing the activity of the `sales_reports` users.

See “Specifying the Application Detail Parameter” on page 24-6 for information on specifying the parameters for `sp_sysmon`.

Sample Output for Application Management

The following sample shows `sp_sysmon` output for the “Application Management” categories in the summary section.

Application Management

Application Statistics Summary (All Applications)

Priority Changes	per sec	per xact	count	% of total
To High Priority	15.7	1.8	5664	49.9 %
To Medium Priority	15.8	1.8	5697	50.1 %
To Low Priority	0.0	0.0	0	0.0 %
Total Priority Changes	31.6	3.5	11361	

Allotted Slices Exhausted	per sec	per xact	count	% of total
High Priority	0.0	0.0	0	0.0 %
Medium Priority	7.0	0.8	2522	100.0 %
Low Priority	0.0	0.0	0	0.0 %
Total Slices Exhausted	7.0	0.8	2522	

Skipped Tasks By Engine	per sec	per xact	count	% of total
Total Engine Skips	0.0	0.0	0	n/a
Engine Scope Changes	0.0	0.0	0	n/a

The following example shows output for application and login; only the information for one application and login is included. Note that the first line identifies the application name (before the arrow) and the login name (after the arrow). Output for application only simply gives the application name at the top of the section.

Application->Login: ctisql->adonis

Application Activity	per sec	per xact	count	% of total
CPU Busy	0.1	0.0	27	2.8 %
I/O Busy	1.3	0.1	461	47.3 %
Idle	1.4	0.2	486	49.9 %
Number of Times Scheduled	1.7	0.2	597	n/a

Application Priority Changes	per sec	per xact	count	% of total
To High Priority	0.2	0.0	72	50.0 %
To Medium Priority	0.2	0.0	72	50.0 %
To Low Priority	0.0	0.0	0	0.0 %
Total Priority Changes	0.4	0.0	144	

Application I/Os Completed	per sec	per xact	count	% of total
Disk I/Os Completed	0.6	0.1	220	53.9 %
Network I/Os Completed	0.5	0.1	188	46.1 %
Total I/Os Completed	1.1	0.1	408	
Resource Limits Violated	per sec	per xact	count	% of total
IO Limit Violations				
Estimated	0.0	0.0	0	0.0 %
Actual	0.1	4.0	4	50.0 %
Time Limit Violations				
Batch	0.0	0.0	0	0.0 %
Xact	0.0	0.0	0	0.0 %
RowCount Limit Violations	0.1	4.0	4	50.0 %
Total Limits Violated	0.1	8.0	8	

Application Statistics Summary (All Applications)

The `sp_sysmon` statistics in the summary section (for all applications) can help you determine whether there are any anomalies in resource utilization. If there are, you can investigate further using the `sp_sysmon applmon` parameter.

This section gives information about:

- Whether tasks are switching back and forth between different priority levels
- Whether the assigned time that tasks are allowed to run is appropriate and whether there are sufficient yield points in the server software
- Whether engine bindings with respect to load balancing is correct

Note that “Application Statistics Summary” includes data for system tasks as well as for user tasks. If the summary report indicates a resource issue, but you do not see supporting evidence in the application or application and login information, investigate the `sp_sysmon` kernel section of the report (“Kernel Utilization” on page 24-10).

Priority Changes

“Priority Changes” reports the priority changes that took place for all user tasks in each priority run queue during the sample interval. It is

normal to see some priority switching due to system-related activity. Such priority switching occurs, for example, when:

- A task sleeps while waiting on a lock—Adaptive Server temporarily raises the task’s priority.
- The housekeeper task sleeps—Adaptive Server raises the priority to medium to sleep, and lowers it back to low when it wakes up.
- A task executes a stored procedure—the task assumes the priority of the stored procedure and resumes its previous priority level after executing the procedure.

This section is useful for verifying that use of the `sp_setpsex` system procedure is not adversely affecting application performance. Compare steady state values before making any user or application priority changes with `sp_setpsex`.

If there are a high number of priority changes with respect to steady state values, it may indicate that an application, or a user task related to that application, is changing priorities frequently. Use the `applmon` option to get priority change data for individual applications. Verify that applications and logins are behaving as you expect.

If you determine that a high priority change rate is not due to an application or to related tasks, then it is likely due to system activity. You might still be able to influence the number of priority changes by using configuration parameters. For example, if the housekeeper task is changing priority frequently, consider resetting the `housekeeper free write percent` configuration parameter.

Total Priority Changes

“Total Priority Changes” reports the total number of priority changes during the sample period. This section gives you a quick way to determine if there are a high number of run queue priority changes occurring.

Allotted Slices Exhausted

“Allotted Slices Exhausted” reports the number of times user tasks in each run queue exceeded the time quantum allotted for execution. Once a user task gains access to an engine, it is allowed to execute for a given **time quantum**. If the task has not yielded the engine before the time quantum is exhausted, Adaptive Server requires it to yield as soon as possible without holding critical resources. After yielding, the task is placed back on the run queue.

This section helps you to determine whether there are CPU-intensive applications for which you should tune execution attributes or engine associations. If these numbers are high, it indicates that an application is CPU intensive. Application-level information can help you figure out which application to tune. Some tasks, especially those which perform large sort operations, are CPU intensive.

Skipped Tasks By Engine

“Skipped Tasks By Engine” reports the number of times engines skipped a user task at the head of a run queue. The value is affected by engine groups and engine group bindings. A high number in this category might be acceptable, but it is possible that an engine group is bound such that a task that is ready to run might not be able to find a compatible engine. In this case, a task might wait to execute while an engine sits idle. Investigate engine groups and how they are bound, and check load balancing.

Engine Scope Changes

“Engine Scope Changes” reports the number of times a user changed the engine group binding of any user task during the sample interval.

Application Statistics per Application or per Application and Login

This section gives detailed information about system resource used by particular application and login tasks, or all users of each application.

Application Activity

“Application Activity” helps you to determine whether an application is I/O intensive or CPU intensive. It reports how much time all user task in the application spend executing, doing I/O, or being idle. It also reports the number of times a task is scheduled and chosen to run.

CPU Busy

“CPU Busy” reports the number of clock ticks during which the user task was executing during the sample interval. When the numbers in

this category are high, it indicates a CPU- bound application. If this is a problem, engine binding might be a solution.

I/O Busy

“I/O Busy” reports the number of clock ticks during which the user task was performing I/O during the sample interval. If the numbers in this category are high, it indicates an I/O-intensive process. If idle time is also high, the application could be disk I/O bound.

The application might achieve better throughput if you assign it a higher priority, bind it to a lightly loaded engine or engine group, or partition the application’s data onto multiple devices.

Idle

“Idle” reports the number of clock ticks during which the user task was idle during the sample interval.

Number of Times Scheduled

“Number of Times Scheduled” reports the number of times a user task is scheduled and chosen to run on an engine. This data can help you determine whether an application has sufficient resources. If this number is low for a task that normally requires substantial CPU time, it may indicate insufficient resources. Consider changing priority in a loaded system with sufficient engine resources.

Application Priority Changes

“Application Priority Changes” reports the number of times this application had its priority changed during the sample interval.

When the “Application Management” category indicates a problem, use this section to pinpoint the source.

Application I/Os Completed

“Application I/Os Completed” reports the disk and network I/Os completed by this application during the sample interval.

This category indicates the total number of disk and network I/Os completed. If you suspect a problem with I/O completion, see “Disk I/O Management” on page 24-86 and “Network I/O Management” on page 24-92.

Resource Limits Violated

“Resource Limits Violated” reports the number and types of violations for:

- I/O Limit Violations–Estimated and Actual
- Time Limits–Batch and Transaction
- RowCount Limit Violations
- “Total Limits Violated”

If no limits are exceeded during the sample period, only the total line is printed.

See Chapter 12, “Limiting Access to Server Resources,” in the *System Administration Guide* for more information on resource limits.

ESP Management

This section reports on the use of extended stored procedures.

Sample Output for ESP Management

ESP Management	per sec	per xact	count	% of total
ESP Requests	0.0	0.0	7	n/a
Avg. Time to Execute an ESP	2.07000 seconds			

ESP Requests

“ESP Requests” reports the number of extended stored procedure calls during the sample interval.

Avg. Time to Execute an ESP

“Avg. Time to Execute an ESP” reports the average length of time for all extended stored procedures executed during the sample interval.

Monitor Access to Executing SQL

This section reports:

- Contention that occurs when `sp_showplan` or Adaptive Server Monitor accesses query plans

- The number of overflows in SQL batch text buffers and the maximum size of SQL batch text sent during the sample interval.

Sample Output for Monitor Access to Executing SQL

Monitor Access to Executing SQL

	per sec	per xact	count	% of total
Waits on Execution Plans	0.1	0.0	5	n/a
Number of SQL Text Overflows	0.0	0.0	1	n/a
Maximum SQL Text Requested (since beginning of sample)	n/a	n/a	4120	n/a

Waits on Execution Plans

“Waits on Execution Plans” reports the number of times that a process attempting to use `sp_showplan` had to wait to acquire read access to the query plan. Query plans may be unavailable if `sp_showplan` is run before the compiled plan is completed or after the query plan finished executing. In these cases, Adaptive Server tries to access the plan three times and then returns a message to the user.

Number of SQL Text Overflows

“Number of SQL Text Overflows” reports the number of times that SQL batch text exceeded the text buffer size.

Maximum SQL Text Requested

“Maximum SQL Text Requested” reports the maximum size of a batch of SQL text since the sample interval began. You can use this value to set the configuration parameter `max SQL text monitored`. See “max SQL text monitored” on page 11-91 of the *System Administration Guide*.

Transaction Profile

This category reports transaction-related activities, including the number of data modification transactions, user log cache (ULC) activity, and transaction log activity.

Sample Output for Transaction Profile

The following sample shows `sp_sysmon` output for the “Transaction Profile” section.

Transaction Profile

```

-----
Transaction Summary      per sec  per xact  count  % of total
-----
Committed Xacts         120.1    n/a      7261   n/a
-----

Transaction Detail      per sec  per xact  count  % of total
-----
Inserts
  Heap Table             120.1    1.0      7260   100.0 %
  Clustered Table        0.0      0.0       0      0.0 %
-----
Total Rows Inserted     120.1    1.0      7260   25.0 %

Updates
  Deferred                0.0      0.0       0      0.0 %
  Direct In-place        360.2    3.0     21774  100.0 %
  Direct Cheap           0.0      0.0       0      0.0 %
  Direct Expensive       0.0      0.0       0      0.0 %
-----
Total Rows Updated      360.2    3.0     21774  75.0 %

Deletes
  Deferred                0.0      0.0       0      0.0 %
  Direct                  0.0      0.0       0      0.0 %
-----
Total Rows Deleted      0.0      0.0       0      0.0 %

```

Transaction Summary

“Transaction Summary” reports committed transactions, rolled back transactions, statistics for all transactions combined, and multidatabase transactions.

Committed Transactions

“Committed Xacts” reports the number of transactions committed during the sample interval. “% of total” reports the percentage of transactions that committed as a percentage of all transactions that started (both committed and rolled back).

This includes transactions that meet explicit, implicit, and ANSI definitions for “committed”, as described here:

- An implicit transaction executes data modification commands such as insert, update, or delete. If you do not specify a begin transaction statement, Adaptive Server interprets every operation as a separate transaction; an explicit commit transaction statement is not required. For example, the following is counted as three transactions.

```
1> insert ...
2> go
1> insert ...
2> go
1> insert ...
2> go
```

- An explicit transaction encloses data modification commands within begin transaction and commit transaction statements and counts the number of transactions by the number of commit statements. For example the following set of statements is counted as one transaction:

```
1> begin transaction
2> insert ...
3> insert ...
4> insert ...
5> commit transaction
6> go
```

- In the ANSI transaction model, any select or data modification command starts a transaction, but a commit transaction statement must complete the transaction. sp_sysmon counts the number of transactions by the number of commit transaction statements. For example, the following set of statements is counted as one transaction:

```
1> insert ...
2> insert ...
3> insert ...
4> commit transaction
5> go
```

If there were transactions that started before the sample interval began and completed during the interval, the value reports a larger number of transactions than the number that started and completed during the sample interval. If transactions do not complete during

the interval, “Total # of Xacts” does not include them. In Figure 24-4, both T1 and T2 are counted, but T3 is not.

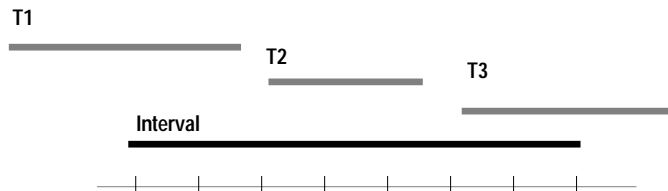


Figure 24-4: How transactions are counted

How to Count Multidatabase Transactions

Multidatabase transactions are also counted. For example, a transaction that modifies three databases is counted as three transactions.

Multidatabase transactions incur more overhead than single database transactions: they require more log records and more ULC flushes, and they involve two-phase commit between the databases.

You can improve performance by reducing the number of multidatabase transactions whenever possible. If you divided a logical database by placing objects in two databases because of contention on the log in SQL Server release 10.0, consider moving objects into the same database.

Transaction Detail

“Transaction Detail” gives statistical detail about data modification operations by type. The work performed by rolled back transactions is included in the output below, although the transaction is not counted in the number of transactions.

See “Update Mode Messages” on page 9-9 for more information on deferred and direct inserts, updates, and deletes.

Inserts

“Inserts” provides detailed information about the types of inserts taking place on heap tables (including partitioned heap tables), clustered tables, and all inserts as a percentage of all insert, update, and delete operations.

This figure does not include fast bulk copy inserts, because those are written directly to the data pages and to disk without the normal insert and logging mechanisms.

Heap Tables

“Heap Tables” reports the number of row inserts that took place on heap tables—all tables that do not have a clustered index. This includes:

- Partitioned heap tables
- Unpartitioned heap tables
- Slow bulk copy inserts into heap tables
- select into commands and inserts into worktables

The “% of total” column shows the percentage of row inserts into heap tables as a percentage of the total number of inserts.

If there are a large number of inserts to heap tables, determine if these inserts are generating contention. Check the `sp_sysmon` report for data on last page locks on heaps in “Lock Detail” on page 24-60. If there appears to be a contention problem, Adaptive Server Monitor can help you figure out which tables are involved.

In many cases, creating a clustered index that randomizes insert activity solves the performance problems for heaps. In other cases, you might need to establish partitions on an unpartitioned table or increase the number of partitions on a partitioned table. For more information, see Chapter 4, “How Indexes Work” and “Improving Insert Performance with Partitions” on page 17-16.

Clustered Tables

“Clustered Table” reports the number of row inserts to tables with clustered indexes. The “% of total” column shows the percentage of row inserts to tables with clustered indexes as a percentage of the total number of rows inserted.

Inserts into clustered tables can lead to page splitting. See “Row ID Updates from Clustered Split” and “Page Splits” on page 24-51.

Total Rows Inserted

“Total Rows Inserted” reports all row inserts to heap tables and clustered tables combined. It gives the average number of all inserts per second, the average number of all inserts per transaction, and the total number of inserts. “% of total” shows the percentage of rows

inserted compared to the total number of rows affected by data modification operations.

Updates

“Updates” reports the number of deferred and direct row updates. The “% of total” column reports the percentage of each type of update as a percentage of the total number of row updates. `sp_sysmon` reports the following types of updates:

- Deferred
- Direct In-place
- Direct Cheap
- Direct Expensive

For a description of update types, see “Optimizing Updates” on page 8-42.

Direct updates incur less overhead than deferred updates and are generally faster because they limit the number of log scans, reduce locking, save traversal of index B-trees (reducing lock contention), and can save I/O because Adaptive Server does not have to refetch pages to perform modification based on log records.

If there is a high percentage of deferred updates, see “Optimizing Updates” on page 8-42.

Total Rows Updated

“Total Rows Updated” reports all deferred and direct updates combined. The “% of total” columns shows the percentage of rows updated, based on all rows modified.

Deletes

“Deletes” reports the number of deferred and direct row deletes. The “% of total” column reports the percentage of each type of delete as a percentage of the total number of deletes. `sp_sysmon` reports deferred and direct deletes.

Total Rows Deleted

“Total Rows Deleted” reports all deferred and direct deletes combined. The “% of total” columns reports the percentage of deleted rows as a compared to all rows inserted, updated, or deleted.

Transaction Management

“Transaction Management” reports transaction management activities, including user log cache (ULC) flushes to transaction logs, ULC log records, ULC semaphore requests, log semaphore requests, transaction log writes, and transaction log allocations.

Sample Output for Transaction Management

The following sample shows `sp_sysmon` output for the “Transaction Management” categories.

Transaction Management

ULC Flushes to Xact Log	per sec	per xact	count	% of total
-----	-----	-----	-----	-----
by Full ULC	0.0	0.0	0	0.0 %
by End Transaction	120.1	1.0	7261	99.7 %
by Change of Database	0.0	0.0	0	0.0 %
by System Log Record	0.4	0.0	25	0.3 %
by Other	0.0	0.0	0	0.0 %
-----	-----	-----	-----	-----
Total ULC Flushes	120.5	1.0	7286	

ULC Log Records	727.5	6.1	43981	n/a
Max ULC Size	n/a	n/a	532	n/a

ULC Semaphore Requests

Granted	1452.3	12.1	87799	100.0 %
Waited	0.0	0.0	0	0.0 %
-----	-----	-----	-----	-----
Total ULC Semaphore Req	1452.3	12.1	87799	

Log Semaphore Requests

Granted	69.5	0.6	4202	57.7 %
Waited	51.0	0.4	3084	42.3 %
-----	-----	-----	-----	-----
Total Log Semaphore Req	120.5	1.0	7286	

Transaction Log Writes	80.5	0.7	4867	n/a
Transaction Log Alloc	22.9	0.2	1385	n/a
Avg # Writes per Log Page	n/a	n/a	3.51408	n/a

ULC Flushes to Transaction Log

“ULC Flushes to Xact Log” reports the total number of times that user log caches (ULCs) were flushed to a transaction log. The “% of total” column reports the percentage of times the type of flush took place, for each category, as a percentage of the total number of ULC flushes. This category can help you identify areas in the application that cause problems with ULC flushes.

There is one user log cache (ULC) for each configured user connection. Adaptive Server uses ULCs to buffer transaction log records. On both SMP and single-processor systems, this helps reduce transaction log I/O. For SMP systems, it reduces the contention on the current page of the transaction log.

You can configure the size of ULCs with the configuration parameter `user log cache size`. See “user log cache size” on page 11-144 of the *System Administration Guide*.

ULC flushes are caused by the following activities:

- “by Full ULC” – a process’s ULC becomes full
- “by End Transaction” – a transaction ended (rollback or commit, either implicit or explicit)
- “by Change of Database” – a transaction modified an object in a different database (a multidatabase transaction)
- “by System Log Record” – a system transaction (such as an OAM page allocation) occurred within the user transaction
- “by Other” – any other reason, including needing to write to disk

When one of these activities causes a ULC flush, Adaptive Server copies all log records from the user log cache to the database transaction log.

“Total ULC Flushes” reports the total number of all ULC flushes that took place during the sample interval.

► **Note**

In databases with mixed data and log segments, the user log cache is flushed after each record is added.

By Full ULC

A high value for “by Full ULC” indicates that Adaptive Server is flushing the ULCs more than once per transaction, negating some performance benefits of user log caches. A good rule of thumb is that if the “% of total” value for “by Full ULC” is greater than 20 percent, consider increasing the size of the user log cache size parameter.

Increasing the ULC size increases the amount of memory required for each user connection, so you do not want to configure the ULC size to suit a small percentage of large transactions.

By End Transaction

A high value for “by End Transaction” indicates a healthy number of short, simple transactions.

By Change of Database

The ULC is flushed every time there is a database change. If this value is high, consider decreasing the size of the ULC if it is greater than 2K. If you divided a logical database by moving tables to a second database because of contention on the log in SQL Server release 10.0, consider moving the tables into one database.

By System Log Record and By Other

If either of these values is higher than approximately 20 percent, and size of your ULC is more than 2048, consider reducing the ULC size.

Check sections of your `sp_sysmon` report that relate to log activity:

- Contention for semaphore on the user log caches (SMP only); see “ULC Semaphore Requests” on page 24-45
- Contention for the log semaphore. (SMP only); see “Log Semaphore Requests” on page 24-46
- The number of transaction log writes; see “Transaction Log Writes” on page 24-47

Total ULC Flushes

“Total ULC Flushes” reports the total number of ULC flushes during the sample interval.

ULC Log Records

This row provides an average number of log records per transaction. It is useful in benchmarking or in controlled development environments to determine the number of log records written to ULCs per transaction.

Many transactions, such as those that affect several indexes or deferred updates or deletes, require several log records for a single data modification. Queries that modify a large number of rows log one or more records for each row.

If this data is unusual, study the data in the next section, “Maximum ULC Size,” and look at your application for long-running transactions and for transactions that modify large numbers of rows.

Maximum ULC Size

The value in the “count” column is the maximum number of bytes used in any ULCs, across all ULCs. This data can help you determine if ULC size is correctly configured.

Since Adaptive Server flushes the ULC when a transaction completes, any unused memory allocated to the ULCs is wasted. If the value in the “count” column is consistently less than the defined value for the user log cache size configuration parameter, reduce user log cache size to the value in the “count” column (but no smaller than 2048 bytes).

When “Max ULC Size” equals the user log cache size, check the number of flushes due to transactions that fill the user log cache (see “By Full ULC” on page 24-44). If the number of times that logs were flushed due to a full ULC is more than 20 percent, consider increasing the user log cache size configuration parameter. See “user log cache size” on page 11-144 in the *System Administration Guide*.

ULC Semaphore Requests

“ULC Semaphore Requests” reports the number of times a user task was immediately granted a semaphore or had to wait for it. “% of total” shows the percentage of tasks granted semaphores and the percentage of tasks that waited for semaphores as a percentage of the total number of ULC semaphore requests. This is relevant only in SMP environments.

A semaphore is a simple internal locking mechanism that prevents a second task from accessing the data structure currently in use. Adaptive Server uses semaphores to protect the user log caches since more than one process can access the records of a ULC and force a flush.

This category provides the following information:

- **Granted** – The number of times a task was granted a ULC semaphore immediately upon request. There was no contention for the ULC.
- **Waited** – The number of times a task tried to write to ULCs and encountered semaphore contention.
- **Total ULC Semaphore Requests** – The total number of ULC semaphore requests that took place during the interval. This includes requests that were granted or had to wait.

Log Semaphore Requests

“Log Semaphore Requests” reports of contention for the log semaphore that protects the current page of the transaction log in cache. This data is meaningful for SMP environments only.

This category provides the following information:

- **Granted** – The number of times a task was granted a log semaphore immediately after it requested one. “% of total” reports the percentage of immediately granted requests as a percentage of the total number of log semaphore requests.
- **Waited** – The number of times two tasks tried to flush ULC pages to the log simultaneously and one task had to wait for the log semaphore. “% of total” reports the percentage of tasks that had to wait for a log semaphore as a percentage of the total number of log semaphore requests.
- **Total Log Semaphore Requests** – The total number of times tasks requested a log semaphore including those granted immediately and those for which the task had to wait.

Log Semaphore Contention and User Log Caches

In high throughput environments with a large number of concurrent users committing transactions, a certain amount of contention for the log semaphore is expected. In some tests, very high throughput is

maintained, even though log semaphore contention is in the range of 20 to 30 percent.

If a high “% of total” for “Waited” shows lot of contention for the log semaphore, some options are:

- Increasing the ULC size, if filling user log caches is a frequent cause of user log cache flushes. See “ULC Flushes to Transaction Log” on page 24-43 for more information.
- Reducing log activity through transaction redesign. Aim for more batching with less frequent commits. Be sure to monitor lock contention as part of the transaction redesign.
- Reducing the number of multidatabase transactions, since each change of database context requires a log write.
- Dividing the database into more than one database so that there are multiple logs. If you choose this solution, divide the database in such a way that multidatabase transactions are minimized.

Transaction Log Writes

“Transaction Log Writes” reports the total number of times Adaptive Server wrote a transaction log page to disk. Transaction log pages are written to disk when a transaction commits (after a wait for a group commit sleep) or when the current log page(s) become full.

Transaction Log Allocations

“Transaction Log Alloc” reports the number of times additional pages were allocated to the transaction log. This data is useful for comparing to other data in this section and for tracking the rate of transaction log growth.

Avg # Writes per Log Page

“Avg # Writes per Log Page” reports the average number of times each log page was written to disk. The value is reported in the “count” column.

In high throughput applications, this number should be as low as possible. If the transaction log uses 2K I/O, the lowest possible value is 1; with 4K log I/O, the lowest possible value is .5, since one log I/O can write 2 log pages.

In low throughput applications, the number will be significantly higher. In very low throughput environments, it may be as high as one write per completed transaction.

Index Management

This category reports index management activity, including nonclustered maintenance, page splits, and index shrinks.

Sample Output for Index Management

The following sample shows `sp_sysmon` output for the "Index Management" categories.

Index Management

Nonclustered Maintenance	per sec	per xact	count	% of total
-----	-----	-----	-----	-----
Ins/Upd Requiring Maint	61.0	4.8	37205	n/a
# of NC Ndx Maint	56.4	4.4	34412	n/a
Avg NC Ndx Maint / Op	n/a	n/a	0.92493	n/a
Deletes Requiring Maint	5.2	0.4	3173	n/a
# of NC Ndx Maint	0.6	0.0	363	n/a
Avg NC Ndx Maint / Op	n/a	n/a	0.11440	n/a
RID Upd from Clust Split	0.0	0.0	0	n/a
# of NC Ndx Maint	0.0	0.0	0	n/a
Avg NC Ndx Maint / Op	0.0	0.0	0	n/a
Page Splits	1.3	0.1	788	n/a
Retries	0.2	0.0	135	17.1 %
Deadlocks	0.0	0.0	14	1.8 %
Empty Page Flushes	0.0	0.0	14	1.8 %
Add Index Level	0.0	0.0	0	0.0 %
Page Shrinks	0.0	0.0	0	n/a
Index Scans	per sec	per xact	count	% of total
-----	-----	-----	-----	-----
Ascending Scans	676.8	16.2	40608	99.7 %
Descending Scans	2.0	0.0	122	0.3 %
-----	-----	-----	-----	-----
Total Scans	678.8	16.2	40730	

Nonclustered Maintenance

This category reports the number of operations that required, or potentially required, maintenance to one or more indexes; that is, it reports the number of operations for which Adaptive Server had to at least check to determine whether it was necessary to update the index. The output also gives the number of indexes that were updated and the average number of indexes maintained per operation.

In tables with clustered indexes and one or more nonclustered indexes, all inserts, all deletes, some update operations, and any data page splits, require changes to the nonclustered indexes. High values for index maintenance indicate that you should assess the impact of maintaining indexes on your Adaptive Server performance. While indexes speed retrieval of data, maintaining indexes slows data modification. Maintenance requires additional processing, additional I/O, and additional locking of index pages.

Other `sp_sysmon` output that is relevant to assessing this category is:

- Information on total updates, inserts and deletes, and information on the number and type of page splits. See “Transaction Detail” on page 24-39, and “Page Splits” on page 24-51.
- Information on lock contention. See “Lock Detail” on page 24-60.
- Information on address lock contention. See “Address Lock Contention” on page 24-25 and “Address Locks” on page 24-61.

For example, you can compare the number of inserts that took place with the number of maintenance operations that resulted. If a relatively high number of maintenance operations, page splits, and retries occurred, consider the usefulness of indexes in your applications. See Chapter 7, “Indexing for Performance,” for more information.

Inserts and Updates Requiring Maintenance to Indexes

The data in this section gives information about how insert and update operations affect indexes. For example, an insert to a clustered table with three nonclustered indexes requires updates to all three indexes, so the average number of operations that resulted in maintenance to nonclustered indexes is three.

However, an update to the same table may require only one maintenance operation—to the index whose key value was changed.

Inserts and Updates Requiring Maintenance

“Ins/Upd Requiring Maint” reports the number of insert and update operations to a table with indexes that potentially required modifications to one or more indexes.

Number of Nonclustered Index Operations Requiring Maintenance

“# of NC Ndx Maint” reports the number of nonclustered indexes that required maintenance as a result of insert and update operations.

Average Number of Nonclustered Indexes Requiring Maintenance

“Avg NC Ndx Maint/Op” reports the average number of nonclustered indexes per insert or update operation that required maintenance.

Deletes Requiring Maintenance

The data in this section gives information about how delete operations affected indexes.

Deletes Requiring Maintenance

“Deletes Requiring Maint” reports the number of delete operations that potentially required modification to one or more indexes. See “Deletes” on page 24-41

Number of Nonclustered Index Operations Requiring Maintenance

“# of NC Ndx Maint” reports the number of nonclustered indexes that required maintenance as a result of delete operations.

Average Number of Nonclustered Indexes Requiring Maintenance

“Avg NC Ndx Maint/Op” reports the average number of nonclustered indexes per delete operation that required maintenance.

Row ID Updates from Clustered Split

This section reports index maintenance activity caused by page splits in tables with clustered indexes. These splits require updating the nonclustered indexes for all of the rows that move to the new data page.

Row ID Updates from Clustered Split

“RID Upd from Clust Split” reports the total number of page splits that required maintenance of a nonclustered index.

Number of Nonclustered Index Operations Requiring Maintenance

“# of NC Ndx Maint” reports the number of nonclustered rows that required maintenance as a result of row ID update operations.

Average Number of Nonclustered Indexes Requiring Maintenance

“Avg NC Ndx Maint/Op” reports the average number of nonclustered indexes entries that were updated for each page split.

Page Splits

“Page Splits” reports the number page splits for data pages, a clustered index pages, or nonclustered index pages because there was not enough room for a new row.

When a data row is inserted into a table with a clustered index, the row must be placed in physical order according to the key value. Index rows must also be placed in physical order on the pages. If there is not enough room on a page for a new row, Adaptive Server splits the page, allocates a new page, and moves some rows to the new page. Page splitting incurs overhead because it involves updating the parent index page and the page pointers on the adjoining pages and adds lock contention. For clustered indexes, page splitting also requires updating all nonclustered indexes that point to the rows on the new page.

See “Choosing Fillfactors for Indexes” on page 7-51 and “Decreasing the Number of Rows per Page” on page 5-35 for more information about how to temporarily reduce page splits using `fillfactor` and `max_rows_per_page`. Note that using `max_rows_per_page` almost always increases the rate of splitting.

Reducing Page Splits for Ascending Key Inserts

If “Page Splits” is high and your application is inserting values into a table with a clustered index on a compound key, it may be possible to reduce the number of page splits through a special optimization that changes the page split point for these indexes.

The special optimization is designed to reduce page splitting and to result in more completely filled data pages. This affects only clustered indexes with compound keys, where the first key is already in use in the table, and the second column is based on an increasing value.

Default Data Page Splitting

The table *sales* has a clustered index on *store_id*, *customer_id*. There are three stores (A, B, and C). Each store adds customer records in ascending numerical order. The table contains rows for the key values A,1; A,2; A,3; B,1; B,2; C,1; C,2; and C,3, and each page holds four rows, as shown in Figure 24-5.

Page 1007			Page 1009		
A	1	...	B	2	...
A	2	...	C	1	...
A	3	...	C	2	...
B	1	...	C	3	...

Figure 24-5: Clustered table before inserts

Using the normal page-splitting mechanism, inserting “A,4” results in allocating a new page and moving half of the rows to it, and inserting the new row in place, as shown in Figure 24-6.

Page 1007			Page 1129			Page 1009		
A	1	...	A	3	...	B	2	...
A	2	...	A	4	...	C	1	...
			B	1	...	C	2	...
						C	3	...

Figure 24-6: Insert causes a page split

When “A,5” is inserted, no split is needed, but when “A,6” is inserted, another split takes place, as shown in Figure 24-7.

Page 1007			Page 1129			Page 1134			Page 1009		
A	1	...	A	3	...	A	5	...	B	2	...
A	2	...	A	4	...	A	6	...	C	1	...
						B	1	...	C	2	...
									C	3	...

Figure 24-7: Another insert causes another page split

Adding “A,7” and “A,8” results in yet another page split, as shown in Figure 24-8.

Page 1007			Page 1129			Page 1134			Page 1137			Page 1009		
A	1	...	A	3	...	A	5	...	A	7	...	B	2	...
A	2	...	A	4	...	A	6	...	A	8	...	C	1	...
									B	1	...	C	2	...
												C	3	...

Figure 24-8: Page splitting continues

Effects of Ascending Inserts

You can set ascending inserts mode for a table, so that pages are split at the point of the inserted row, rather than in the middle of the page. Starting from the original table shown in Figure 24-5 on page 24-52, the insertion of “A,4” results in a split at the insertion point, with the remaining rows on the page moving to a newly allocated page, as shown in Figure 24-9.

Page 1007			Page 1129			Page 1009		
A	1	...	B	1	...	B	2	...
A	2	...				C	1	...
A	3	...				C	2	...
A	4	...				C	3	...

Figure 24-9: First insert with ascending inserts mode

Inserting “A,5” causes a new page to be allocated, as shown in Figure 24-10.

Page 1007			Page 1134			Page 1129			Page 1009		
A	1	...	A	5	...	B	1	...	B	2	...
A	2	...							C	1	...
A	3	...							C	2	...
A	4	...							C	3	...

Figure 24-10: Additional ascending insert causes a page allocation

Adding “A,6”, “A,7”, and “A,8” fills the new page, as shown in Figure 24-11.

Page 1007			Page 1134			Page 1129			Page 1009		
A	1	...	A	5	...	B	1	...	B	2	...
A	2	...	A	6	...				C	1	...
A	3	...	A	7	...				C	2	...
A	4	...	A	8	...				C	3	...

Figure 24-11: Additional inserts fill the new page

Setting Ascending Inserts Mode for a Table

The following command turns on ascending insert mode for the *sales* table:

```
dbcc tune (ascinserts, 1, "sales")
```

To turn ascending insert mode off, use:

```
dbcc tune (ascinserts, 0, "sales")
```

These commands update the *status2* bit of *sysindexes*.

If tables sometimes experience random inserts and have more ordered inserts during batch jobs, it is better to enable `dbcc tune (ascinserts)` only for the period during which the batch job runs.

Retries and Deadlocks

“Deadlocks” reports the number of index page splits and shrinks that resulted in deadlocks. Adaptive Server has a special mechanism called **deadlock retries** that attempts to avoid transaction rollbacks caused by index page deadlocks. “Retries” reports the number of times Adaptive Server used this mechanism.

Deadlocks on index pages take place when each of two transactions needs to acquire locks held by the other transaction. On data pages, deadlocks result in choosing one process (the one with the least accumulated CPU time) as a deadlock victim and rolling back the process.

However, by the time that an index deadlock takes place, the transaction has already found and updated the data page and is holding data page locks. Rolling back the transaction causes overhead.

In a large percentage of index deadlocks caused by page splits and shrinks, both transactions can succeed by dropping one set of index

locks, and restarting the index scan. The index locks for one of the processes are released (locks on the data pages are still held), and Adaptive Server tries the index scan again, traversing the index from the root page of the index.

Usually, by the time the scan reaches the page that needs to be split, the other transaction has completed, and no deadlock takes place. By default, any index deadlock that is due to a page split or shrink is retried up to five times before the transaction is considered deadlocked and is rolled back. For information on changing the default value for the number of deadlock retries, see “deadlock retries” on page 11-53 of the *System Administration Guide*.

The deadlock retries mechanism causes the locks on data pages to be held slightly longer than usual and causes increased locking and overhead. However, it reduces the number of transactions that are rolled back due to deadlocks. The default setting provides a reasonable compromise between the overhead of holding data page locks longer and the overhead of rolling back transactions that have to be reissued.

A high number of index deadlocks and deadlock retries indicates high contention in a small area of the index B-tree.

If your application encounters a high number of deadlock retries, reduce page splits using `fillfactor` when you re-create the index. See “Decreasing the Number of Rows per Page” on page 5-35.

Empty Page Flushes

“Empty Page Flushes” reports the number of empty pages resulting from page splits that were flushed to disk.

Add Index Level

“Add Index Level” reports the number of times a new index level was added. This does not happen frequently, so you should expect to see result values of 0 most of the time. The count could have a value of 1 or 2 if your sample includes inserts into either an empty table or a small table with indexes.

Page Shrinks

“Page Shrinks” reports the number of times that deleting index rows caused the index to shrink off a page. Shrinks incur overhead due to

locking in the index and the need to update pointers on adjacent pages. Repeated “count” values greater than 0 indicate there may be many pages in the index with fairly small numbers of rows per page due to delete and update operations. If there are a high number of shrinks, consider rebuilding the indexes.

Index Scans

This section prints the total number of scans performed and the number of ascending and descending scans.

Metadata Cache Management

“Metadata Cache Management” reports the use of the metadata caches that store information about the three types of metadata caches: objects, indexes, and databases. This section also reports the number of object, index and database descriptors that were active during the sample interval, and the maximum number of descriptors that have been used since the server was last started. It also reports spinlock contention for the object and index metadata caches.

Sample Output for Metadata Cache Management

Metadata Cache Management

```

-----
Metadata Cache Summary      per sec   per xact   count   % of total
-----
Open Object Usage
Active                      0.4       0.1       116     n/a
Max Ever Used Since Boot   0.4       0.1       121     n/a
Free                       1.3       0.3       379     n/a
Reuse Requests
  Succeeded                 0.0       0.0        0       n/a
  Failed                   0.0       0.0        0       n/a

Open Index Usage
Active                      0.2       0.1        67     n/a
Max Ever Used Since Boot   0.2       0.1        72     n/a
Free                       1.4       0.3       428     n/a
Reuse Requests
  Succeeded                 0.0       0.0        0       n/a
  Failed                   0.0       0.0        0       n/a

Open Database Usage

```

Active	0.0	0.0	10	n/a
Max Ever Used Since Boot	0.0	0.0	10	n/a
Free	0.0	0.0	2	n/a
Reuse Requests				
Succeeded	0.0	0.0	0	n/a
Failed	0.0	0.0	0	n/a
Object Spinlock Contention	n/a	n/a	n/a	0.0 %
Index Spinlock Contention	n/a	n/a	n/a	1.0 %
Hash Spinlock Contention	n/a	n/a	n/a	1.0 %

Open Object, Index, and Database Usage

Each of these sections contains the same information for the three types of metadata caches. The output provides this information:

- “Active” reports the number of objects, indexes, or databases that were active during the sample interval.
- “Max Ever Used Since Boot” reports the maximum number of descriptors used since the last restart of Adaptive Server.
- “Free” reports the number of free descriptors in the cache.
- “Reuse Requests” reports the number of times that the cache had to be searched for reusable descriptors:
 - “Failed” means that all descriptors in cache were in use and that the client issuing the request received an error message.
 - “Succeeded” means the request found a reusable descriptor in cache. Even though “Succeeded” means that the client did not get an error message, Adaptive Server is doing extra work to locate reusable descriptors in the cache and to read metadata information from disk.

You can use this information to set the configuration parameters **number of open indexes**, **number of open objects**, and **number of open databases**, as shown in Table 24-2.

Table 24-2: Action to take based on metadata cache usage statistics

<i>sp_sysmon</i> Output	Action
Large number of “Free” descriptors	Set parameter lower
Very few “Free” descriptors	Set parameter higher
“Reuse Requests Succeeded” nonzero	Set parameter higher
“Reuse Requests Failed” nonzero	Set parameter higher

Open Object and Open Index Spinlock Contention

These sections report on contention for the spinlocks on the object descriptor and index descriptor caches. You can use this information to tune the configuration parameters **open object spinlock ratio** and **open index spinlock ratio**. If the reported contention is more than 3 percent, decrease the value of the corresponding parameter to lower the number of objects or indexes that are protected by a single spinlock.

Open Index Hash Spinlock Contention

This section reports contention for the spinlock on the index metadata cache hash table. You can use this information to tune the **open index hash spinlock ratio** configuration parameter. If the reported contention is greater than 3 percent, decrease the value of the parameter.

Lock Management

“Lock Management” reports locks, deadlocks, lock promotions, and lock contention.

Sample Output for Lock Management

The following sample shows *sp_sysmon* output for the “Lock Management” categories.

Lock Management

Lock Summary	per sec	per xact	count	% of total
Total Lock Requests	2540.8	21.2	153607	n/a
Avg Lock Contention	3.7	0.0	224	0.1 %
Deadlock Percentage	0.0	0.0	0	0.0 %

Lock Detail	per sec	per xact	count	% of total
-------------	---------	----------	-------	------------

Exclusive Table

Granted	403.7	4.0	24376	100.0 %
Waited	0.0	0.0	0	0.0 %
Total EX-Table Requests	0.0	0.0	0	0.0 %

Shared Table

Granted	325.2	4.0	18202	100.0 %
Waited	0.0	0.0	0	0.0 %
Total SH-Table Requests	0.0	0.0	0	0.0 %

Exclusive Intent

Granted	480.2	4.0	29028	100.0 %
Waited	0.0	0.0	0	0.0 %
Total EX-Intent Requests	480.2	4.0	29028	18.9 %

Shared Intent

Granted	120.1	1.0	7261	100.0 %
Waited	0.0	0.0	0	0.0 %
Total SH-Intent Requests	120.1	1.0	7261	4.7 %

Exclusive Page

Granted	483.4	4.0	29227	100.0 %
Waited	0.0	0.0	0	0.0 %
Total EX-Page Requests	483.4	4.0	29227	19.0 %

Update Page

Granted	356.5	3.0	21553	99.0 %
Waited	3.7	0.0	224	1.0 %

Total UP-Page Requests	360.2	3.0	21777	14.2 %
Shared Page				
Granted	3.2	0.0	195	100.0 %
Waited	0.0	0.0	0	0.0 %
-----	-----	-----	-----	-----
Total SH-Page Requests	3.2	0.0	195	0.1 %
Exclusive Address				
Granted	134.2	1.1	8111	100.0 %
Waited	0.0	0.0	0	0.0 %
-----	-----	-----	-----	-----
Total EX-Address Requests	134.2	1.1	8111	5.3 %
Shared Address				
Granted	959.5	8.0	58008	100.0 %
Waited	0.0	0.0	0	0.0 %
-----	-----	-----	-----	-----
Total SH-Address Requests	959.5	8.0	58008	37.8 %
Last Page Locks on Heaps				
Granted	120.1	1.0	7258	100.0 %
Waited	0.0	0.0	0	0.0 %
-----	-----	-----	-----	-----
Total Last Pg Locks	120.1	1.0	7258	4.7 %
Deadlocks by Lock Type				
-----	per sec	per xact	count	% of total
-----	-----	-----	-----	-----
Total Deadlocks	0.0	0.0	0	n/a
Deadlock Detection				
Deadlock Searches	0.1	0.0	4	n/a
Searches Skipped	0.0	0.0	0	0.0 %
Avg Deadlocks per Search	n/a	n/a	0.00000	n/a
Lock Promotions				
Total Lock Promotions	0.0	0.0	0	n/a

“Lock Promotions” does report detail rows if there were no occurrences of them during the sample interval. In this sample report, “Deadlocks by Lock Type” is one example.

Lock Summary

“Lock Summary” provides overview statistics about lock activity that took place during the sample interval.

Total Lock Requests

“Total Lock Requests” reports the total number of lock requests.

Average Lock Contention

“Avg Lock Contention” reports the average number of times there was lock contention as a percentage of the total number of locks requests.

If the lock contention average is high, study the lock detail information below. See “Locking and Performance” on page 5-32 for more information on tuning locking behavior.

Deadlock Percentage

“Deadlock Percentage” reports the percentage of deadlocks as a percentage of the total number lock requests. If this value is high, see “Deadlocks by Lock Type” on page 24-62.

Lock Detail

“Lock Detail” provides information that you can use to determine whether the application is causing a lock contention or deadlock-related problem.

This output reports locks by type, displaying the number of times that each lock type was granted immediately, and the number of times a task had to wait for a particular type of lock. The “% of total” is the percentage of the specific lock type that was granted or had to wait with respect to the total number of lock requests.

“Lock Detail” reports the following types of locks:

- Exclusive Table
- Shared Table
- Exclusive Intent
- Shared Intent
- Exclusive Page

- Update Page
- Shared Page
- Exclusive Address
- Shared Address
- Last Page Locks on Heaps

Lock contention can have a large impact on Adaptive Server performance. Table locks generate more lock contention than page locks because no other tasks can access a table while there is an exclusive table lock on it, and if a task requires an exclusive table lock, it must wait until all shared locks are released.

You can try redesigning the tables that have the highest lock contention or the queries that acquire and hold the locks, to reduce the number of locks they hold and the length of time the locks are held. Table, page, and intent locks are described in “Types of Locks in Adaptive Server” on page 5-3.

Address Locks

“Exclusive Address” and “Shared Address” report the number of times address locks were granted immediately or the number of times the task had to wait for the lock. Address locks are held on index pages.

Last Page Locks on Heaps

“Last Page Locks on Heaps” reports locking attempts on the last page of a partitioned or unpartitioned heap table.

This information can indicate whether there are tables in the system that would benefit from partitioning or from increasing the number of partitions. Adding a clustered index that distributes inserts randomly across the data pages may also help. If you know that one or more tables is experiencing a problem with contention for the last page, Adaptive Server Monitor can help determine which table is experiencing the problem.

See “Improving Insert Performance with Partitions” on page 17-16 for information on how partitions can help solve the problem of last-page locking on unpartitioned heap tables.

Deadlocks by Lock Type

“Deadlocks by Lock Type” reports the number of specific types of deadlocks. “% of total” gives the number of each deadlock type as a percentage of the total number of deadlocks.

Deadlocks may occur when many transactions execute at the same time in the same database. They become more common as the lock contention increases between the transactions.

This category reports data for the following deadlock types:

- Exclusive Table
- Shared Table
- Exclusive Intent
- Shared Intent
- Exclusive Page
- Update Page
- Shared Page
- Address

“Total Deadlocks” summarizes the data for all lock types.

As in the example for this section, if there are no deadlocks, `sp_sysmon` does not display any detail information, it only prints the “Total Deadlocks” row with zero values.

To pinpoint where deadlocks occur, try running several applications in a controlled environment with deadlock information printing enabled. See “print deadlock information” on page 11-122 in the *System Administration Guide*.

For more information on lock types, see “Types of Locks in Adaptive Server” on page 5-3.

For more information on deadlocks and coping with lock contention, see “Deadlocks and Concurrency” on page 5-28 and “Locking and Performance” on page 5-32.

Deadlock Detection

“Deadlock Detection” reports the number of deadlock searches that found deadlocks and deadlock searches that were skipped during the sample interval.

For a discussion of the background issues related to this topic, see “Deadlocks and Concurrency” on page 5-28.

Deadlock Searches

“Deadlock Searches” reports the number of times that Adaptive Server initiated a deadlock search during the sample interval.

Deadlock checking is time-consuming overhead for applications that experience no deadlocks or very low levels of deadlocking. You can use this data with “Average Deadlocks per Search” to determine if Adaptive Server is checking for deadlocks too frequently.

Searches Skipped

“Searches Skipped” reports the number of times that a task started to perform deadlock checking, but found deadlock checking in progress and skipped its check. “% of total” reports the percentage of deadlock searches that were skipped as a percentage of the total number of searches.

When a process is blocked by lock contention, it waits for an interval of time set by the configuration parameter `deadlock checking period`. When this period elapses, it starts deadlock checking. If a search is already in process, the process skips the search.

If you see some number of searches skipped, but some of the searches are finding deadlocks, increase the parameter slightly. If you see a lot of searches skipped, and no deadlocks, or very few, you can increase the parameter by a larger amount.

See “deadlock checking period” on page 11-52 in the *System Administration Guide* for more information.

Average Deadlocks per Search

“Avg Deadlocks per Search” reports the average number deadlocks found per search.

This category measures whether Adaptive Server is checking too frequently. For example, you might decide that finding one deadlock per search indicates excessive checking. If so, you can adjust the frequency with which tasks search for deadlocks by increasing the value of the `deadlock checking period` configuration parameter. See “deadlock checking period” on page 11-52 in the *System Administration Guide* for more information.

Lock Promotions

“Lock Promotions” reports the number of times that the following escalations took place:

- “Ex-Page to Ex-Table” – exclusive page to exclusive table
- “Sh-Page to Sh-Table” – shared page to shared table

The “Total Lock Promotions” row reports the average number of lock promotion types combined per second and per transaction.

If no lock promotions took place during the sample interval, only the total row is printed.

If there are no lock promotions, `sp_sysmon` does not display the detail information, as the example for this section shows.

“Lock Promotions” data can:

- Help you detect if lock promotion in your application to is a cause of lock contention and deadlocks
- Be used before and after tuning lock promotion variables to determine the effectiveness of the values.

Look at the “Granted” and “Waited” data above for signs of contention. If lock contention is high and lock promotion is frequent, consider changing the lock promotion thresholds for the tables involved.

You can configure the lock promotion threshold either server-wide or for individual tables. See “How Isolation Levels Affect Locking” on page 5-12.

Data Cache Management

`sp_sysmon` reports summary statistics for all caches followed by statistics for each named cache.

`sp_sysmon` reports the following activities for the default data cache and for each named cache:

- Spinlock contention
- Utilization
- Cache searches including hits and misses
- Pool turnover for all configured pools
- Buffer wash behavior, including buffers passed clean, buffers already in I/O, and buffers washed dirty

- Prefetch requests performed and denied
- Dirty read page requests

Figure 24-12 shows how these caching features relate to disk I/O and the data caches.

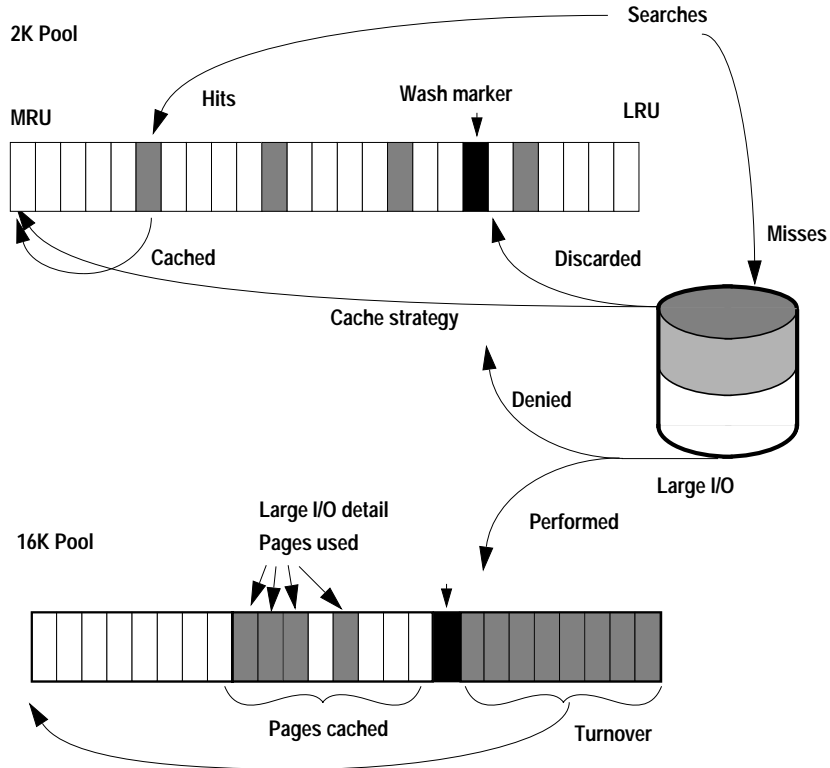


Figure 24-12: Cache management categories

You can use the system procedures `sp_cacheconfig` and `sp_helpcache` output to help you analyze the data from this section of the report. `sp_cacheconfig` provides information about caches and pools, and `sp_helpcache` provides information about objects bound to caches. See Chapter 9, “Configuring Data Caches,” in the *System Administration Guide* for information on how to use these system procedures. See “Named Data Caches” on page 16-12 for more information on performance issues and named caches.

Sample Output for Data Cache Management

The following sample shows `sp_sysmon` output for the "Data Cache Management" categories. The first block of data, "Cache Statistics Summary," includes information for all caches. The output also reports a separate block of data for each cache. These blocks are identified by the cache name. The sample output shown here includes only a single user defined cache, although there were more caches configured during the interval.

Data Cache Management

Cache Statistics Summary (All Caches)

	per sec	per xact	count	% of total
Cache Search Summary				
Total Cache Hits	7520.5	524.7	1804925	99.3 %
Total Cache Misses	55.9	3.9	13411	0.7 %
Total Cache Searches	7576.4	528.6	1818336	
Cache Turnover				
Buffers Grabbed	47.1	3.3	11310	n/a
Buffers Grabbed Dirty	0.0	0.0	0	0.0 %
Cache Strategy Summary				
Cached (LRU) Buffers	6056.0	422.5	1453437	99.8 %
Discarded (MRU) Buffers	11.4	0.8	2734	0.2 %
Large I/O Usage				
Large I/Os Performed	7.3	0.5	1752	49.1 %
Large I/Os Denied	7.6	0.5	1819	50.9 %
Total Large I/O Requests	14.9	1.0	3571	
Large I/O Effectiveness				
Pages by Lrg I/O Cached	55.9	3.9	13424	n/a
Pages by Lrg I/O Used	43.6	3.0	10475	78.0 %
Asynchronous Prefetch Activity				
APFs Issued	9.3	0.6	2224	30.1 %
APFs Denied Due To				
APF I/O Overloads	0.2	0.0	36	0.5 %
APF Limit Overloads	0.7	0.0	158	2.1 %
APF Reused Overloads	0.4	0.0	100	1.4 %
APF Buffers Found in Cache				
With Spinlock Held	0.0	0.0	1	0.0 %
W/o Spinlock Held	20.3	1.4	4865	65.9 %

Total APFs Requested	30.8	2.1	7384	
Other Asynchronous Prefetch Statistics				
APFs Used	8.7	0.6	1819	n/a
APF Waits for I/O	4.0	0.3	965	n/a
APF Discards	0.0	0.0	0	n/a
Dirty Read Behavior				
Page Requests	0.0	0.0	0	n/a

Cache: default data cache				
	per sec	per xact	count	% of total
Spinlock Contention	n/a	n/a	n/a	24.0 %
Utilization	n/a	n/a	n/a	93.4 %
Cache Searches				
Cache Hits	7034.6	490.8	1688312	99.4 %
Found in Wash	2.4	0.2	583	0.0 %
Cache Misses	42.7	3.0	10250	0.6 %

Total Cache Searches	7077.3	493.8	1698562	
Pool Turnover				
2 Kb Pool				
LRU Buffer Grab	30.7	2.1	7371	82.0 %
Grabbed Dirty	0.0	0.0	0	0.0 %
16 Kb Pool				
LRU Buffer Grab	6.7	0.5	1616	18.0 %
Grabbed Dirty	0.0	0.0	0	0.0 %

Total Cache Turnover	37.4	2.6	8987	
Buffer Wash Behavior				
Buffers Passed Clean	0.3	0.0	64	100.0 %
Buffers Already in I/O	0.0	0.0	0	0.0 %
Buffers Washed Dirty	0.0	0.0	0	0.0 %
Cache Strategy				
Cached (LRU) Buffers	5571.9	388.7	1337248	99.8 %
Discarded (MRU) Buffers	11.4	0.8	2732	0.2 %
Large I/O Usage				
Large I/Os Performed	6.7	0.5	1614	47.1 %
Large I/Os Denied	7.6	0.5	1814	52.9 %

Total Large I/O Requests	14.3	1.0	3428	
Large I/O Detail				
16 Kb Pool				
Pages Cached	53.9	3.8	12928	n/a
Pages Used	42.4	3.0	10173	78.7 %
Dirty Read Behavior				
Page Requests	0.0	0.0	0	n/a

Cache Statistics Summary (All Caches)

This section summarizes behavior for the default data cache and all named data caches combined. Corresponding information is printed for each data cache. For a full discussion of the rows in this section, see “Cache Management By Cache” on page 24-74.

Cache Search Summary

This section provides summary information about cache hits and misses. Use this data to get an overview of how effective cache design is. A high number of cache misses indicates that you should investigate statistics for each cache.

Total Cache Hits

“Total Cache Hits” reports the number of times that a needed page was found in any cache. “% of total” reports the percentage of cache hits as a percentage of the total number of cache searches.

Total Cache Misses

“Total Cache Misses” reports the number of times that a needed page was not found in a cache and had to be read from disk. “% of total” reports the percentage of times that the buffer was not found in the cache as a percentage of all cache searches.

Total Cache Searches

“Total Cache Searches” reports the total number of cache searches, including hits and misses for all caches combined.

Cache Turnover

This section provides a summary of cache turnover.

Buffers Grabbed

“Buffers Grabbed” reports the number of buffers that were replaced in all of the caches. The “count” column represents the number of times that Adaptive Server fetched a buffer from the LRU end of the cache, replacing a database page. If the server was recently restarted, so that the buffers are empty, reading a page into an empty buffer is not counted here.

Buffers Grabbed Dirty

“Buffers Grabbed Dirty” reports the number of times that fetching a buffer found a dirty page at the LRU end of the cache and had to wait while the buffer was written to disk. If this value is nonzero, find out which caches are affected. It represents a serious performance hit.

Cache Strategy Summary

This section provides a summary of the caching strategy used.

Cached (LRU) Buffers

“Cached (LRU) Buffers” reports the total number of buffers placed at the head of the MRU/LRU chain in all caches.

Discarded (MRU) Buffers

“Discarded (MRU) Buffers” reports the total number of buffers in all caches following the fetch-and-discard strategy—the buffers placed at the wash marker.

Large I/O Usage

This section provides summary information about the large I/O requests in all caches. If “Large I/Os Denied” is high, investigate individual caches to determine the cause.

Large I/Os Performed

“Large I/Os Performed” measures the number of times that the requested large I/O was performed. “% of total” is the percentage of

large I/O requests performed as a percentage of the total number of I/O requests made.

Large I/Os Denied

“Large I/Os Denied” reports the number of times that large I/O could not be performed. “% of total” reports the percentage of large I/O requests denied as a percentage of the total number of requests made.

Total Large I/O Requests

This row reports the number of all large I/O requests (both granted and denied) for all caches.

Large I/O Effectiveness

“Large I/O Effectiveness” helps you to determine the performance benefits of large I/O. It compares the number of pages that were brought into cache by a large I/O to the number of pages actually referenced while in the cache. If the percentage for “Pages by Lrg I/O Used” is low, it means that few of the pages brought into cache are being accessed by queries. Investigate the individual caches to determine the source of the problem.

Pages by Lrg I/O Cached

“Pages by Lrg I/O Cached” reports the number of pages brought into all caches by all the large I/O operations that took place during the sample interval.

Low percentages could indicate one of the following:

- Allocation fragmentation in the table’s storage
- Inappropriate caching strategy

Pages by Lrg I/O Used

“Pages by Lrg I/O Used” reports the total number of pages that were used after being brought into cache as part of a large I/O. `sp_sysmon` does not print output for this category if there were no “Pages by Lrg I/O Cached.”

Asynchronous Prefetch Activity Report

This section reports asynchronous prefetch activity for all caches. For information on asynchronous prefetch for each database device, see “Disk I/O Management” on page 24-86.

“Total APFs Requested” reports the total number of pages eligible to be prefetched, that is, the sum of the look-ahead set sizes of all queries issued during the sample interval. Other rows in “Asynchronous Prefetch Activity” provide detail in the three following categories:

- Information about the pages that were prefetched, “APFs Issued”
- Information about the reasons that prefetch was denied
- Information about how the page was found in the cache

APFs Issued

“APFs Issued” reports the number of asynchronous prefetch requests issued by the system during the sample interval.

APFs Denied Due To

This section reports the reasons that APFs were not issued:

- “APF I/O Overloads” reports the number of times APF usage was denied because of a lack of disk I/O structures or because of disk semaphore contention.

If this number is high, check the following information in the “Disk I/O Management” section of the report:

- Check the value of the `disk i/o structures` configuration parameter. See “Disk I/O Structures” on page 24-89.
- Check values for contention for device semaphores for each database device to determine the source of the problem. See “Device Semaphore Granted and Waited” on page 24-92 for more information.

If the problem is due to lack of a disk I/O structures, set the configuration parameter higher, and repeat your tests. If the problem is due to high disk semaphore contention, examine the physical placement of the objects where high I/O takes place.

- “APF Limit Overloads” indicates that the percentage of buffer pools that can be used for asynchronous prefetch was exceeded. This limit is set for the server as a whole by the `global async prefetch`

limit configuration parameter. It can be tuned for each pool with `sp_poolconfig`.

- “APF Reused Overloads” indicates that APF usage was denied due to a kinked page chain or because the buffers brought in by APF were swapped out before they could be accessed.

APF Buffers Found in Cache

This section reports how many buffers from APF look-ahead sets were found in the data cache during the sample interval.

Asynchronous prefetch tries to find a page it needs to read in the data cache using a quick scan without holding the cache spinlock. If that does not succeed, it then performs a thorough scan holding the spinlock.

Other Asynchronous Prefetch Statistics

Three additional asynchronous prefetch statistics are reported in this section:

- “APFs Used” reports the number of pages that were brought into the cache by asynchronous prefetch and used during the sample interval. The pages counted for this report may have been brought into cache during the sample interval or by asynchronous prefetch requests that were issued before the sample interval started.
- “APF Waits for I/O” reports the number of times that a process had to wait for an asynchronous prefetch to complete. This indicates that the prefetch was not issued early enough for the pages to be in cache before the query needed them. It is reasonable to expect some percentage of “APF Waits”. Some reasons that tasks may have to wait are:
 - The first asynchronous prefetch request for a query is generally included in “APF Waits”.
 - Each time a sequential scan moves to a new allocation unit and issues prefetch requests, the query must wait until the first I/O completes.
 - Each time a nonclustered index scan finds a set of qualified rows and issues prefetch requests for the pages, it must wait for the first pages to be returned.

Other factors that can affect “APF Waits for I/O” are the amount of processing that needs to be done on each page and the speed of the I/O subsystem.

- “APF Discards” indicates the number of pages read in by asynchronous prefetch and discarded before they were used. A high value for “APFs Discards” may indicate that increasing the size of the buffer pools could help performance, or it may indicate that APF is bringing pages into cache that are not needed by the query.

Dirty Read Behavior

This section provides information to help you analyze how dirty reads (isolation level 0 reads) affect the system.

Page Requests

“Page Requests” reports the average number of pages that were requested at isolation level 0.

The “% of total” column reports the percentage of dirty reads with respect to the total number of page reads.

Dirty read page requests incur high overhead if they lead to many dirty read restarts. Therefore, the dirty read page request data is most valuable when you use it with the data for “Dirty Read Re-Starts”.

Dirty Read Re-Starts

“Re-Starts” reports the number of dirty read restarts that took place. This category is reported only for the server as a whole, and not for individual caches. `sp_sysmon` does not print output for this category if there were no “Dirty Read Page Requests,” as in the sample output.

A dirty read restart occurs when a dirty read is active on a page and another process makes changes to the page that cause the page to be deallocated. The scan for the level 0 must be restarted.

The “% of total” output is the percentage of dirty read restarts done with isolation level 0 as a percentage of the total number of page reads.

If these values are high, you might take steps to reduce them through application modifications because overhead associated with dirty reads and resulting restarts is very expensive. Most applications should avoid restarts because of the large overhead it incurs.

Cache Management By Cache

This sections reports cache utilization for each active cache on the server. The sample output shows results for the default data cache. The following section explains the per-cache statistics.

Spinlock Contention

“Spinlock Contention” reports the number of times an engine encountered spinlock contention on the cache, and had to wait, as a percentage of the total spinlock requests for that cache. This is meaningful for SMP environments only.

When a user task makes any changes to a cache, a spinlock denies all other tasks access to the cache while the changes are being made. Although spinlocks are held for extremely brief durations, they can slow performance in multiprocessor systems with high transaction rates. If spinlock contention is more than 15 percent, consider using named caches.

To improve performance, you can divide the default data cache into named data caches. Each cache is protected by a separate spinlock. This can increase concurrency on multiple CPU systems. See “Named Data Caches” on page 16-12.

Utilization

“Utilization” reports the percentage of searches that went to the cache in question as a percentage of searches across all caches configured.

You can compare this value for each cache to determine if there are caches that are over- or under-utilized. If you decide that a cache is not well utilized, you can:

- Change the cache bindings to balance utilization. For more information, see “Caches and Object Bindings” on page 3-16 and “Binding Objects to Caches” on page 9-15 in the *System Administration Guide* for more information.
- Resize the cache to correspond more appropriately to its utilization. For more information, see “Resizing Named Data Caches” on page 9-24 in the *System Administration Guide*.

Cache Search, Hit, and Miss Information

This section displays the number hits and misses and the total number of searches for this cache. Cache hits are roughly comparable to the logical reads values reported by `statistics io`; cache misses are roughly equivalent to physical reads. `sp_sysmon` always reports values that are higher than those shown by `statistics io`, since `sp_sysmon` also reports the I/O for system tables, log pages, OAM pages and other system overhead.

Interpreting cache hit data requires an understanding of how the application uses each cache. In caches that are created to hold specific objects such as indexes or look up tables, cache hit ratios may reach 100 percent. In caches used for random point queries on huge tables, cache hit ratios may be quite low but still represent effective cache use.

This data can also help you to determine if adding more memory would improve performance. For example, if “Cache Hits” is high, adding memory probably would not help much.

Cache Hits

“Cache Hits” reports the number of times that a needed page was found in the data cache. “% of total” reports the percentage of cache hits compared to the total number of cache searches.

Found in Wash

The number of times that the needed page was found in the wash section of the cache. “% of total” reports the percentage of times that the buffer was found in the wash area as a percentage of the total number of hits.

If the data indicate a large percentage of cache hits found in the wash section, it may mean the wash is too big. Of course, it is not a problem for caches that are read-only or that have a low number of writes.

A large wash section might lead to increased physical I/O because Adaptive Server initiates a write on all dirty pages as they cross the wash marker. If a page in the wash area is written to disk, then updated a second time, I/O has been wasted. Check to see whether a large number of buffers are being written at the wash marker. See “Buffers Washed Dirty” on page 24-79 for more information.

If queries on tables in the cache use “fetch-and-discard” strategy for a non-APF I/O, the first cache hit for a page finds it in the wash. The buffers is moved to the MRU end of the chain, so a second cache hit

soon after the first cache hit will find the buffer still outside the wash area. See “Discarded (MRU) Buffers” on page 24-79 for more information, and “Specifying the Cache Strategy” on page 10-13 for information about controlling caching strategy.

If necessary, you can change the wash size. See “Changing the Wash Area for a Memory Pool” on page 9-20 for more information. If you make the wash size smaller, run `sp_sysmon` again under fully loaded conditions and check the output for “Grabbed Dirty” values greater than 0. See “Buffers Grabbed Dirty” on page 24-69.

Cache Misses

“Cache Misses” reports the number of times that a needed page was not found in the cache and had to be read from disk. “% of total” is the percentage of times that the buffer was not found in the cache as a percentage of the total searches.

Total Cache Searches

This row summarizes cache search activity. Note that the “Found in Wash” data is a subcategory of the “Cache Hits” number and it is not used in the summary calculation.

Pool Turnover

“Pool Turnover” reports the number of times that a buffer is replaced from each pool in a cache. Each cache can have up to 4 pools, with I/O sizes of 2K, 4K, 8K, and 16K. If there is any “Pool Turnover,” `sp_sysmon` prints the “LRU Buffer Grab” and “Grabbed Dirty” information for each pool that is configured and a total turnover figure for the entire cache. If there is no “Pool Turnover,” `sp_sysmon` prints only a row of zeros for “Total Cache Turnover.”

This information helps you to determine if the pools and cache are the right size.

LRU Buffer Grab

“LRU Buffer Grab” is incremented only when a page is replaced by another page. If you have recently restarted Adaptive Server, or if you have just unbound and rebound the object or database to the cache, turnover does not count reading pages into empty buffers.

If memory pools are too small for the throughput, you may see high turnover in the pools, reduced cache hit rates, and increased I/O rates. If turnover is high in some pools and low in other pools, you

might want to move space from the less active pool to the more active pool, especially if it can improve the cache-hit ratio.

If the pool has 1000 buffers, and Adaptive Server is replacing 100 buffers every second, 10 percent of the buffers are being turned over every second. That might be an indication that the buffers do not remain in cache for long enough for the objects using that cache.

Grabbed Dirty

“Grabbed Dirty” gives statistics for the number of dirty buffers that reached the LRU before they could be written to disk. When Adaptive Server needs to grab a buffer from the LRU end of the cache in order to fetch a page from disk, and finds a dirty buffer instead of a clean one, it must wait for I/O on the dirty buffer to complete. “% of total” reports the percentage of buffers grabbed dirty as a percentage of the total number of buffers grabbed.

If “Grabbed Dirty” is a nonzero value, it indicates that the wash area of the pool is too small for the throughput in the pool. Remedial actions depend on the pool configuration and usage:

- If the pool is very small and has high turnover, consider increasing the size of the pool and the wash area.
- If the pool is large, and it is used for a large number of data modification operations, increase the size of the wash area.
- If several objects use the cache, moving some of them to another cache could help.
- If the cache is being used by create index, the high I/O rate can cause dirty buffer grabs, especially in a small 16K pool. In these cases, set the wash size for the pool as high as possible, to 80 percent of the buffers in the pool.
- Check query plans and I/O statistics for objects that use the cache for queries that perform a lot of physical I/O in the pool. Tune queries, if possible, by adding indexes.

Check the “per second” values for “Buffers Already in I/O” on page 24-78 and “Buffers Washed Dirty” on page 24-79. The wash area should be large enough to allow I/O to be completed on dirty buffers before they reach the LRU. The time required to complete the I/O depends on the actual number of physical writes per second achieved by your disk drives.

Also check “Disk I/O Management” on page 24-86 to see if I/O contention is slowing disk writes.

Also, it might help to increase the value of the **housekeeper free write percent** configuration parameter. See “housekeeper free write percent” on page 11-104 in the *System Administration Guide*.

Total Cache Turnover

This summary line provides the total number of buffers grabbed in all pools in the cache.

Buffer Wash Behavior

This category reports information about the state of buffers when they reach the pool's wash marker. When a buffer reaches the wash marker it can be in one of three states:

- **Clean** – the buffer was not changed while it was in the cache, or it was changed, and has already been written to disk by the housekeeper or a checkpoint. When the write completes, the page remains in cache and is marked clean.
- **Already in I/O** – the page was dirtied while in the cache. The housekeeper or a checkpoint has started I/O on the page, but the I/O has not completed.
- **Dirty** – the buffer was changed while in the cache and has not been written to disk. An asynchronous I/O is started on the page as it passes the wash marker.

If no buffers pass the wash marker during the sample interval, `sp_sysmon` prints:

```
Statistics Not Available - No Buffers Entered Wash  
Section Yet!
```

Buffers Passed Clean

“Buffers Passed Clean” reports the number of buffers that were clean when they passed the wash marker. “% of total” reports the percentage of buffers passed clean as a percentage of the total number of buffers that passed the wash marker.

Buffers Already in I/O

“Buffers Already in I/O” reports the number of times that I/O was already active on a buffer when it entered the wash area. “% of total” reports the percentage of buffers already in I/O as a percentage of the total number of buffers that entered the wash area.

If I/Os are active on pages as they cross the wash marker, it is because the housekeeper task or the checkpoint process started the I/O. See “housekeeper free write percent” on page 11-104 in the *System Administration Guide* for more information about configuring the housekeeper.

Buffers Washed Dirty

“Buffers Washed Dirty” reports the number of times that a buffer entered the wash area dirty and not already in I/O. “% of total” reports the percentage of buffers washed dirty as a percentage of the total number of buffers that entered the wash area.

Cache Strategy

This section reports the number of buffers placed in cache following the fetch-and-discard (MRU) or normal (LRU) caching strategies.

Cached (LRU) Buffers

“Cached(LRU) Buffers” reports the number of buffers that used normal cache strategy and were placed at the MRU end of the cache. This includes all buffers read directly from disk and placed at the MRU end, and all buffers that were found in cache. At the completion of the logical I/O, the buffer was placed at the MRU end of the cache.

Discarded (MRU) Buffers

“Discarded (MRU) Buffers” reports the number of buffers that were placed at the wash marker, using the fetch-and-discard strategy.

If you expect an entire table to be cached, but you see a high value for “Discarded Buffers,” use `showplan` to see if the optimizer is generating the fetch-and-discard strategy when it should be using the normal cache strategy. See “Specifying the Cache Strategy” on page 10-13 for more information.

Large I/O Usage

This section provides data about Adaptive Server prefetch requests for large I/O. It reports statistics on the numbers of large I/O requests performed and denied.

Large I/Os Performed

“Large I/Os Performed” measures the number of times that a requested large I/O was performed. “% of total” reports the percentage of large I/O requests performed as a percentage of the total number of requests made.

Large I/Os Denied

“Large I/Os Denied” reports the number of times that large I/O could not be performed. “% of total” reports the percentage of large I/O requests denied as a percentage of the total number of requests made.

Adaptive Server cannot perform large I/O:

- If any page in a buffer already resides in another pool.
- When there are no buffers available in the requested pool.
- On the first extent of an allocation unit, since it contains the allocation page, which is always read into the 2K pool. This means that on a large table scan, at least one large I/O out of 32 will be denied.

If a high percentage of large I/Os were denied, it indicates that the use of the larger pools might not be as effective as it could be. If a cache contains a large I/O pool, and queries perform both 2K and 16K I/O on the same objects, there will always be some percentage of large I/Os that cannot be performed because pages are in the 2K pool.

If more than half of the large I/Os were denied, and you are using 16K I/O, try moving all of the space from the 16K pool to the 8K pool. Rerun the test to see if total I/O is reduced. Note that when a 16K I/O is denied, Adaptive Server does not check for 8K or 4K pools, but uses the 2K pool.

You can use information from this category and “Pool Turnover” to help judge the correct size for pools.

Total Large I/O Requests

“Total Large I/O Requests” provides summary statistics for large I/Os performed and denied for all caches combined.

Large I/O Detail

This section provides summary information for each pool individually. It contains a block of information for each 4K, 8K, or 16K pool configured in cache. It prints the pages brought in (“Pages Cached”) and pages referenced (“Pages Used”) for each I/O size that is configured.

For example, if a query performs a 16K I/O and reads a single data page, the “Pages Cached” value is 8, and “Pages Used” value is 1.

Pages Cached

“Pages by Lrg I/O Cached” prints the total number of pages read into the cache.

Pages Used

“Pages by Lrg I/O Used” reports the number of pages used by a query while in cache.

Dirty Read Behavior

“Page Requests” reports the average number of pages that were requested at isolation level 0.

The “% of total” output for “Dirty Read Page Requests” shows the percentage of dirty reads with respect to the total number of page reads.

Procedure Cache Management

“Procedure Cache Management” reports the number of times stored procedures and triggers were requested, read from disk, and removed.

Sample Output for Procedure Cache Management

The following sample shows `sp_sysmon` output for the “Procedure Cache Management” section.

Procedure Cache Management	per sec	per xact	count	% of total
Procedure Requests	67.7	1.0	4060	n/a
Procedure Reads from Disk	0.0	0.0	0	0.0 %
Procedure Writes to Disk	0.0	0.0	0	0.0 %
Procedure Removals	0.0	0.0	0	n/a

Procedure Requests

“Procedure Requests” reports the number of times stored procedures were executed.

When a procedure is executed, these possibilities exist:

- An idle copy of the query plan in memory, so it is copied and used.
- No copy of the procedure is in memory, or all copies of the plan in memory are in use, so the procedure must be read from disk.

Procedure Reads from Disk

“Procedure Reads from Disk” reports the number of times that stored procedures were read from disk rather than copied in the procedure cache.

“% of total” reports the percentage of procedure reads from disk as a percentage of the total number of procedure requests. If this is a relatively high number, it could indicate that the procedure cache is too small.

Procedure Writes to Disk

“Procedure Writes to Disk” reports the number of procedures created during the interval. This can be significant if application programs generate stored procedures.

Procedure Removals

“Procedure Removals” reports the number of times that a procedure aged out of cache.

Memory Management

“Memory Management” reports the number of pages allocated and deallocated during the sample interval.

Sample Output for Memory Management

The following sample shows `sp_sysmon` output for the “Memory Management” section.

Memory Management	per sec	per xact	count	% of total
Pages Allocated	0.0	0.0	0	n/a
Pages Released	0.0	0.0	0	n/a

Pages Allocated

“Pages Allocated” reports the number of times that a new page was allocated in memory.

Pages Released

“Pages Released” reports the number of times that a page was freed.

Recovery Management

This data indicates the number of checkpoints caused by the normal checkpoint process, the number of checkpoints initiated by the housekeeper task, and the average length of time for each type. This information is helpful for setting the recovery and housekeeper parameters correctly.

Sample Output for Recovery Management

The following sample shows `sp_sysmon` output for the “Recovery Management” section.

Recovery Management

Checkpoints	per sec	per xact	count	% of total
# of Normal Checkpoints	0.00117	0.00071	1	n/a
# of Free Checkpoints	0.00351	0.00213	3	n/a
Total Checkpoints	0.00468	0.00284	4	
Avg Time per Normal Chkpt	0.01050 seconds			
Avg Time per Free Chkpt	0.16221 seconds			

Checkpoints

Checkpoints write all dirty pages (pages that have been modified in memory, but not written to disk) to the database device. Adaptive Server's automatic (normal) checkpoint mechanism works to maintain a minimum recovery interval. By tracking the number of log records in the transaction log since the last checkpoint was performed, it estimates whether the time required to recover the transactions exceeds the recovery interval. If so, the checkpoint process scans all caches and writes all changed data pages to the database device.

When Adaptive Server has no user tasks to process, a housekeeper task automatically begins writing dirty buffers to disk. These writes are done during the server's idle cycles, so they are known as "free writes." They result in improved CPU utilization and a decreased need for buffer washing during transaction processing.

If the housekeeper process finishes writing all dirty pages in all caches to disk, it checks the number of rows in the transaction log since the last checkpoint. If there are more than 100 log records, it issues a checkpoint. This is called a "free checkpoint" because it requires very little overhead. In addition, it reduces future overhead for normal checkpoints.

Number of Normal Checkpoints

"# of Normal Checkpoints" reports the number of checkpoints caused by the normal checkpoint process.

If the normal checkpoint is doing most of the work, especially if the time required is lengthy, it might make sense to increase the number of writes performed by the housekeeper task.

See “recovery interval in minutes” on page 11-22 and “Synchronizing a Database and Its Log: Checkpoints” on page 20-3 in the *System Administration Guide* for information about changing the number of normal checkpoints.

Number of Free Checkpoints

“# of Free Checkpoints” reports the number of checkpoints initiated by the housekeeper task. The housekeeper performs checkpoints to the log only when it has cleared all dirty pages from all configured caches.

If the housekeeper is doing most of the checkpoints, you can probably increase the recovery interval without affecting performance or actual recovery time. Increasing the recovery interval reduces the number of normal checkpoints and the overhead incurred by them.

You can use the `housekeeper free write percent` parameter to configure the maximum percentage by which the housekeeper task can increase database writes. For more information about configuring the housekeeper task, see “housekeeper free write percent” on page 11-104 in the *System Administration Guide*.

Total Checkpoints

“Total Checkpoints” reports the combined number of normal and free checkpoints that occurred during the sample interval.

Average Time per Normal Checkpoint

“Avg Time per Normal Chkpt” reports the time, on average during the sample interval, that normal checkpoints lasted.

Average Time per Free Checkpoint

“Avg Time per Free Chkpt” reports the time, on average during the sample interval, that free (or housekeeper) checkpoints lasted.

Increasing the Housekeeper Batch Limit

The housekeeper process has a built-in batch limit to avoid overloading disk I/O for individual devices. By default, the batch

size for housekeeper writes is set to 3. As soon as the housekeeper detects that it has issued 3 I/Os to a single device, it stops processing in the current buffer pool and begins checking for dirty pages in another pool. If the writes from the next pool need to go to the same device, it continues to another pool. Once the housekeeper has checked all of the pools, it waits until the last I/O it has issued has completed, and then begins the cycle again.

The default batch limit of 3 is designed to provide good device I/O characteristics for slow disks. You may get better performance by increasing the batch size for fast disk drives. This limit can be set globally for all devices on the server or to different values for disks with different speeds. You must reset the limits each time Adaptive Server is restarted.

This command sets the batch size to 10 for a single device, using the virtual device number from `sysdevices`:

```
dbcc tune(deviochar, 8, "10")
```

To see the device number, use `sp_helpdevice` or this query:

```
select name, low/16777216
from sysdevices
where status&2=2
```

To change the housekeeper's batch size for all devices on the server, use -1 in place of a device number:

```
dbcc tune(deviochar, -1, "5")
```

Legal values for the batch size are 1–255. For very fast drives, setting the batch size as high as 50 has yielded performance improvements during testing.

You may want to try setting the batch size higher if:

- The average time for normal checkpoints is high
- There are no problems with exceeding I/O configuration limits or contention on the semaphores for the devices

Disk I/O Management

This section reports on disk I/O. It provides an overview of disk I/O activity for the server as a whole and reports on reads, writes, and semaphore contention for each logical device.

Sample Output for Disk I/O Management

The following sample shows sp_sysmon output for the “Disk I/O Management” section.

Disk I/O Management

```

-----
Max Outstanding I/Os      per sec  per xact  count  % of total
-----
Server                    n/a      n/a      74     n/a
Engine 0                  n/a      n/a      20     n/a
Engine 1                  n/a      n/a      21     n/a
Engine 2                  n/a      n/a      18     n/a
Engine 3                  n/a      n/a      23     n/a
Engine 4                  n/a      n/a      18     n/a
Engine 5                  n/a      n/a      20     n/a
Engine 6                  n/a      n/a      21     n/a
Engine 7                  n/a      n/a      17     n/a
Engine 8                  n/a      n/a      20     n/a

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

Total Requested Disk I/Os  202.8    1.7    12261   n/a

Completed Disk I/O's
Engine 0                  25.0    0.2    1512    12.4 %
Engine 1                  21.1    0.2    1274    10.5 %
Engine 2                  18.4    0.2    1112     9.1 %
Engine 3                  23.8    0.2    1440    11.8 %
Engine 4                  22.7    0.2    1373    11.3 %
Engine 5                  22.9    0.2    1387    11.4 %
Engine 6                  24.4    0.2    1477    12.1 %
Engine 7                  22.0    0.2    1332    10.9 %
Engine 8                  21.2    0.2    1281    10.5 %
-----
Total Completed I/Os      201.6    1.7    12188

```

Device Activity Detail

```

-----
/dev/rdisk/clt3d0s6
tpcd_data5          per sec  per xact  count  % of total
-----
Reads
  APF                0.7      1395    75.7 %
  Non-APF            0.1       286    15.5 %
Writes
  APF                0.1       162     8.8 %
-----
Total I/Os          0.9      1843    6.1 %

Device Semaphore Granted  0.9      1839    99.6 %
Device Semaphore Waited   0.0         8     0.4 %

Device Semaphore Granted  80.7         0.7    4878    100.0 %
Device Semaphore Waited   0.0         0.0         0     0.0 %
-----

d_master
master          per sec  per xact  count  % of total
-----
Reads
  APF            56.6         0.5    3423    46.9 %
  Non-APF        64.2         0.5    3879    53.1 %
Writes
  APF            64.2         0.5    3879    53.1 %
-----
Total I/Os      120.8         1.0    7302    60.0 %

Device Semaphore Granted  116.7         1.0    7056    94.8 %
Device Semaphore Waited   6.4         0.1     388     5.2 %

```

Maximum Outstanding I/Os

“Max Outstanding I/Os” reports the maximum number of I/Os pending for Adaptive Server as a whole (the first line), and for each Adaptive Server engine at any point during the sample interval.

This information can help configure I/O parameters at the server or operating system level if any of the “I/Os Delayed By” values are nonzero.

I/Os Delayed By

When the system experiences an I/O delay problem, it is likely that I/O is blocked by one or more Adaptive Server or operating system limits.

Most operating systems have a kernel parameter that limits the number of asynchronous I/Os that can take place.

Disk I/O Structures

“Disk I/O Structures” reports the number of I/Os delayed by reaching the limit on disk I/O structures. When Adaptive Server exceeds the number of available disk I/O control blocks, I/O is delayed because Adaptive Server requires that tasks get a disk I/O control block before initiating an I/O request.

If the result is a nonzero value, try increasing the number of available disk I/O control blocks by increasing the configuration parameter `disk i/o structures`. See “disk i/o structures” on page 11-37 in the *System Administration Guide*.

Server Configuration Limit

Adaptive Server can exceed its limit for the number of asynchronous disk I/O requests that can be outstanding for the entire Adaptive Server at one time. You can raise this limit using the `max async i/os per server` configuration parameter. See “max async i/os per server” on page 11-81 in the *System Administration Guide*.

Engine Configuration Limit

An engine can exceed its limit for outstanding asynchronous disk I/O requests. You can change this limit with the `max async i/os per engine` configuration parameter. See “max async i/os per engine” on page 11-81 in the *System Administration Guide*.

Operating System Limit

“Operating System Limit” reports the number of times the operating system limit on outstanding asynchronous I/Os was exceeded during the sample interval. The operating system kernel limits the maximum number of asynchronous I/Os that either a process or the entire system can have pending at any one time. See “disk i/o

structures” on page 11-37 in the *System Administration Guide*; also see your operating system documentation.

Requested and Completed Disk I/Os

This data shows the total number of disk I/Os requested by Adaptive Server, and the number and percentage of I/Os completed by each Adaptive Server engine.

“Total Requested Disk I/Os” and “Total Completed I/Os” should be the same or very close. These values will be very different if requested I/Os are not completing due to saturation.

The value for requested I/Os includes all requests that were initiated during the sample interval, and it is possible that some of them completed after the sample interval ended. These I/Os will not be included in “Total Completed I/Os”, and will cause the percentage to be less than 100, when there are no saturation problems.

The reverse is also true. If I/O requests were made before the sample interval began and they completed during the period, you would see a “% of Total” for “Total Completed I/Os” value that is more than 100 percent. If you are checking for saturation, make repeated runs, and try to develop your stress tests to perform relatively consistent levels of I/O.

If the data indicates a large number of requested disk I/Os and a smaller number of completed disk I/Os, there could be a bottleneck in the operating system that is delaying I/Os.

Total Requested Disk I/Os

“Total Requested Disk I/Os” reports the number of times that Adaptive Server requested disk I/Os.

Completed Disk I/Os

“Total Completed Disk I/Os” reports the number of times that each engine completed I/O. “% of total” reports the percentage of times each engine completed I/Os as a percentage of the total number of I/Os completed by all Adaptive Server engines combined.

You can also use this information to determine whether the operating system can keep pace with the disk I/O requests made by all of the engines.

Device Activity Detail

“Device Activity Detail” reports activity on the master device and on each logical device. It is useful for checking that I/O is well balanced across the database devices and for finding a device that might be delaying I/O. For example, if the “Task Context Switches Due To” data indicates a heavy amount of device contention, you can use “Device Activity Detail” to figure out which device(s) is causing the problem.

This section prints the following information about I/O for each data device on the server:

- The logical and physical device names
- The number of reads and writes and the total number of I/Os
- The number of device semaphore requests immediately granted on the device and the number of times a process had to wait for a device semaphore

Reads and Writes

“Reads” and “Writes” report the number of times that reads or writes to a device took place. “Reads” reports the number of pages that were read by asynchronous prefetch and those brought into cache by other I/O activity. The “% of total” column reports the percentage of reads or writes as a percentage of the total number of I/Os to the device.

Total I/Os

“Total I/Os” reports the combined number of reads and writes to a device. The “% of total” column is the percentage of combined reads and writes for each named device as a percentage of the number of reads and writes that went to all devices.

You can use this information to check I/O distribution patterns over the disks and to make object placement decisions that can help balance disk I/O across devices. For example, does the data show that some disks are more heavily used than others? If you see that a large percentage of all I/O went to a specific named device, you can investigate the tables residing on the device and then determine how to remedy the problem. See “Creating Objects on Segments” on page 17-11.

Device Semaphore Granted and Waited

The “Device Semaphore Granted” and “Device Semaphore Waited” categories report the number of times that a request for a device semaphore was granted immediately and the number of times the semaphore was busy and the task had to wait for the semaphore to be released. The “% of total” column is the percentage of times the device the semaphore was granted (or the task had to wait) as a percentage of the total number of device semaphores requested. This data is meaningful for SMP environments only.

When Adaptive Server needs to perform a disk I/O, it gives the task the semaphore for that device in order to acquire a block I/O structure. This is important on SMP systems, because it is possible to have multiple Adaptive Server engines trying to post I/Os to the same device simultaneously. This creates contention for that semaphore, especially if there are hot devices or if the data is not well distributed across devices.

A large percentage of I/O requests that waited could indicate a semaphore contention issue. One solution might be to redistribute the data on the physical devices.

Network I/O Management

“Network I/O Management” reports the following network activities for each Adaptive Server engine:

- Total requested network I/Os
- Network I/Os delayed
- Total TDS packets and bytes received and sent
- Average size of packets received and sent

This data is broken down by engine, because each engine does its own networking. Imbalances are usually caused by one of the following condition:

- There are more engines than tasks, so the engines with no work to perform report no I/O, or
- Most tasks are sending and receiving short packets, but another task is performing heavy I/O, such as a bulk copy.

Sample Output for Network I/O Management

The following sample shows sp_sysmon output for the “Network I/O Management” categories.

Network I/O Management

```
-----
Total Network I/O Requests      240.1      2.0      14514      n/a
  Network I/Os Delayed          0.0        0.0         0         0.0 %
```

```
-----
Total TDS Packets Received      per sec    per xact    count    % of total
-----
  Engine 0                       7.9         0.1         479      6.6 %
  Engine 1                       12.0        0.1         724      10.0 %
  Engine 2                       15.5        0.1         940      13.0 %
  Engine 3                       15.7        0.1         950      13.1 %
  Engine 4                       15.2        0.1         921      12.7 %
  Engine 5                       17.3        0.1        1046      14.4 %
  Engine 6                       11.7        0.1         706       9.7 %
  Engine 7                       12.4        0.1         752      10.4 %
  Engine 8                       12.2        0.1         739      10.2 %
-----
```

```
Total TDS Packets Rec'd          120.0      1.0       7257
```

```
-----
Total Bytes Received            per sec    per xact    count    % of total
-----
  Engine 0                     562.5         4.7      34009      6.6 %
  Engine 1                     846.7         7.1     51191     10.0 %
  Engine 2                    1100.2         9.2     66516     13.0 %
  Engine 3                    1112.0         9.3     67225     13.1 %
  Engine 4                    1077.8         9.0     65162     12.7 %
  Engine 5                    1219.8        10.2     73747     14.4 %
  Engine 6                     824.3         6.9     49835      9.7 %
  Engine 7                     879.2         7.3     53152     10.4 %
  Engine 8                     864.2         7.2     52244     10.2 %
-----
```

```
Total Bytes Rec'd              8486.8     70.7    513081
```

```
Avg Bytes Rec'd per Packet      n/a        n/a         70      n/a
```

```
-----
Total TDS Packets Sent          per sec    per xact    count    % of total
-----
  Engine 0                       7.9         0.1         479      6.6 %
  Engine 1                       12.0        0.1         724      10.0 %
  Engine 2                       15.6        0.1         941      13.0 %
  Engine 3                       15.7        0.1         950      13.1 %
```

Engine 4	15.3	0.1	923	12.7 %
Engine 5	17.3	0.1	1047	14.4 %
Engine 6	11.7	0.1	705	9.7 %
Engine 7	12.5	0.1	753	10.4 %
Engine 8	12.2	0.1	740	10.2 %

Total TDS Packets Sent	120.1	1.0	7262	
Total Bytes Sent	per sec	per xact	count	% of total

Engine 0	816.1	6.8	49337	6.6 %
Engine 1	1233.5	10.3	74572	10.0 %
Engine 2	1603.2	13.3	96923	13.0 %
Engine 3	1618.5	13.5	97850	13.1 %
Engine 4	1572.5	13.1	95069	12.7 %
Engine 5	1783.8	14.9	107841	14.4 %
Engine 6	1201.1	10.0	72615	9.7 %
Engine 7	1282.9	10.7	77559	10.4 %
Engine 8	1260.8	10.5	76220	10.2 %

Total Bytes Sent	12372.4	103.0	747986	
Avg Bytes Sent per Packet	n/a	n/a	103	n/a

Total Network I/Os Requests

“Total Network I/O Requests” reports the total number of TDS packets received and sent.

If you know how many packets per second the network can handle, you can determine whether Adaptive Server is challenging the network bandwidth.

The issues are the same whether the I/O is inbound or outbound. If Adaptive Server receives a command that is larger than the TDS packet size, Adaptive Server will wait to begin processing until it receives the full command. Therefore, commands that require more than one packet are slower to execute and take up more I/O resources.

If the average bytes per packet is near the default packet size configured for your server, you may want to configure larger packet sizes for some connections. You can configure the network packet size for all connections or allow certain connections to log in using larger packet sizes. See “Changing Network Packet Sizes” on page 20-3 in the *System Administration Guide*.

Network I/Os Delayed

“Network I/Os Delayed” reports the number of times I/O was delayed. If this number is consistently nonzero, consult with your network administrator.

Total TDS Packets Received

“Total TDS Packets Received” reports the number of TDS packets received per engine. “Total TDS Packets Rec’d” reports the number of packets received during the sample interval.

Total Bytes Received

“Total Bytes Received” reports the number of bytes received per engine. “Total Bytes Rec’d” reports the total number of bytes received during the sample interval.

Average Bytes Received per Packet

“Average Bytes Rec’d per Packet” reports the average number of bytes for all packets received during the sample interval.

Total TDS Packets Sent

“Total TDS Packets Sent” reports the number of packets sent by each engine, and a total for the server as a whole.

Total Bytes Sent

“Total Bytes Sent” reports the number of bytes sent by each Adaptive Server engine, and the server as a whole, during the sample interval.

Average Bytes Sent per Packet

“Average Bytes Sent per Packet” reports the average number of bytes for all packets sent during the sample interval.

Reducing Packet Overhead

If your applications use stored procedures, you may see improved throughput by turning off certain TDS messages that are sent after each select statement that is performed in a stored procedure. This message, called a “done in proc” message, is used in some client products. In some cases, turning off “done in proc” messages also turns off the “rows returned” messages. These messages may be expected in certain Client-Library programs, but many clients simply discard these results. Test the setting with your client products and Open Client programs to determine whether it affects them before disabling this message on a production system.

Turning off “done in proc” messages can increase throughput slightly in some environments, especially those with slow or overloaded networks, but may have virtually no effect in other environments. To turn the messages off, issue the command:

```
dbcc tune (doneinproc, 0)
```

To turn the messages on, use:

```
dbcc tune (doneinproc, 1)
```

This command must be issued each time Adaptive Server is restarted.

Index

The index is divided into three sections:

- Symbols
Indexes each of the symbols used in this guide.
- Numerics
Indexes entries that begin numerically.
- Subjects
Indexes subjects alphabetically.

Page numbers in **bold** are primary references.

Symbols

- > (greater than)
optimizing 11-1
- # (pound sign), temporary table
identifier prefix 19-2

Numerics

- 16K memory pool, estimating I/O
for 6-9
- 302 trace flag 10-17 to 10-29
- 4K memory pool, transaction log
and 16-30
- 8K memory pool, uses of 16-16

A

Access

- See also* Access methods
- index 3-3
- memory and disk speeds 16-1
- optimizer methods 3-2, 14-4 to 14-12
- Access methods 14-4
 - hash-based 14-4
 - hash-based scan 14-4

- optimizer choices 8-4
- parallel 14-4 to 14-13
- partition-based 14-4
- selection of 14-12
- showplan** messages for **9-23 to 9-39**
- Add index level, **sp_sysmon** report 24-56
- Address locks
 - contention 24-26
 - deadlocks reported by
sp_sysmon 24-64
 - sp_sysmon** report on 24-63
- Affinity
 - CPU 21-11, 21-21
 - engine example 22-11
- Aggregate functions
 - denormalization and
performance 2-11
 - denormalization and temporary
tables 19-5
 - ONCE AGGREGATE messages in
showplan 9-55
 - optimization of 8-26, 11-5
 - parallel optimization of 14-29
 - showplan** messages for 9-14
 - subqueries including 8-30
- Aging

- data cache 16-8
- procedure cache 16-4
- all keyword
 - union, optimization of 11-5
- Allocation map. *See* Object Allocation Map (OAM)
- Allocation pages **3-8**
 - large I/O and 24-82
- Allocation units 3-5, 3-8
 - database creation and 23-1
 - dbcc report on 6-9
- alter table command
 - parallel sorting and 15-9
- and keyword
 - subqueries containing 8-31
- any keyword
 - subquery optimization and 8-28
- Application design 24-4
 - cursors and 12-16
 - deadlock avoidance 5-30
 - deadlock detection in 5-30
 - delaying deadlock checking 5-31
 - denormalization for 2-9
 - DSS and OLTP 16-13
 - index specification 10-9
 - isolation level 0 considerations 5-14
 - levels of locking 5-37
 - managing denormalized data
 - with 2-16, 2-17
 - network packet size and 20-5
 - network traffic reduction with 20-7
 - primary keys and 7-33
 - procedure cache sizing 16-6
 - SMP servers 21-23
 - temporary tables in 19-5
 - user connections and 24-23
 - user interaction in transactions 5-33
- Application execution precedence 22-1 to 22-29
 - environment analysis 22-28
 - scheduling and 22-10
 - system procedures 22-7
 - tuning with `sp_sysmon` 24-30
- Application queues. *See* Application execution precedence
- Applications
 - CPU usage report 24-34
 - disk I/O report 24-35
 - I/O usage report 24-35
 - idle time report 24-35
 - network I/O report 24-35
 - priority changes 24-35
 - TDS messages and 24-98
 - yields (CPU) 24-35
- Architecture
 - multithreaded 21-1
 - Server SMP 21-20
- Arguments, search. *See* Search arguments
- Artificial columns 7-51
- Ascending scan `showplan` message 9-28
- Ascending sort 7-27, 7-29
- ascinserts (dbcc tune parameter) 24-55
- Assigning execution precedence 22-2
- Asynchronous I/O
 - buffer wash behavior and 24-80
 - `sp_sysmon` report on 24-91
 - statistics io report on 7-11
- Asynchronous prefetch **18-1 to 18-16**
 - dbcc and 18-5, 18-14
 - denied due to limits 24-73
 - during recovery 18-3
 - fragmentation and 18-8
 - hash-based scans and 18-13
 - large I/O and 18-11
 - look-ahead set 18-2
 - maintenance for 18-15
 - MRU replacement strategy and 18-12
 - nonclustered indexes and 18-4
 - page chain fragmentation and 18-8
 - page chain kinks and 18-8, 18-15
 - parallel query processing and 18-12
 - partition-based scans and 18-13
 - performance monitoring 18-16
 - pool limits and 18-7
 - recovery and 18-14
 - sequential scans and 18-4

- Cache hit ratio
 - cache replacement policy and 16-27
 - data cache 16-10
 - partitioning and 13-23
 - procedure cache 16-6
 - sp_sysmon report on 24-70, 24-77
- Cache replacement policy 16-19 to 16-23
 - comparison of types 16-22
 - defined 16-19
 - indexes 16-34
 - lookup tables 16-34
 - transaction logs 16-34
- Caches, data 16-7 to 16-39
 - aging in 3-16
 - binding objects to 3-16
 - cache hit ratio 16-10
 - data modification and 3-19, 16-9
 - deletes on heaps and 3-20
 - fillfactor and 16-39
 - guidelines for named 16-27
 - hits found in wash 24-77
 - hot spots bound to 16-12
 - I/O configuration 3-15, 16-16 to 16-17
 - inserts to heaps and 3-19
 - joins and 3-18
 - large I/O and 16-13
 - misses 24-78
 - MRU replacement strategy 3-18
 - named 16-12 to 16-35
 - page aging in 16-8
 - parallel sorting and 15-11
 - pools in 3-15, 16-16 to 16-17
 - sorts and 15-11 to 15-12
 - spinlocks on 16-12
 - strategies chosen by optimizer 16-18, 24-81
 - subquery results 8-32
 - table scans and 7-17
 - task switching and 24-24
 - tempdb bound to own 16-13, 19-10
 - total searches 24-78
 - transaction log bound to own 16-13
 - updates to heaps and 3-20
 - utilization 24-76
 - wash marker 3-16
- Canceling
 - queries with adjusted plans 14-32
- Case sensitivity
 - in SQL 1
- Chain of buffers (data cache) 3-16
- Chains of pages
 - data and index 3-5
 - index pages 4-4
 - overflow pages and 4-10
 - partitions 17-14
 - placement 17-1
 - unpartitioning 17-23
- Changing
 - configuration parameters 24-3
- Character data
 - joins and compatibility 11-7
- Character expressions li
 - char datatype
 - null becomes varchar 11-7
- Cheap direct updates 8-36
- checkalloc option, dbcc
 - object size report 6-7
- Checkpoint process 16-8, 24-86
 - average time 24-87
 - CPU usage 24-14
 - housekeeper task and 21-17
 - I/O batch size 24-25
 - sp_sysmon and 24-85
- Client
 - connections 21-1
 - packet size specification 20-5
 - service request 21-16
 - task 21-2
 - TDS messages 24-98
- Client/server architecture 20-3
- close command
 - memory and 12-5
- close on endtran option, set 12-16
- Clustered indexes **4-2**
 - asynchronous prefetch and scans 18-4
 - computing number of pages 6-15, 6-16
 - computing size of rows 6-15

- create index requirements 15-8
- dbcc and size of 6-8
- delete operations 4-10
- estimating size of 6-14 to 6-25
- fillfactor effect on 6-25
- guidelines for choosing 7-31
- insert operations and 4-7
- order of key values 4-4
- overflow pages and 4-10
- overhead 3-21
- page reads 4-6
- page splits and 24-52
- partitioned tables and 17-24
- performance and 3-21
- point query cost 7-20
- prefetch and 10-11
- range query cost 7-20
- reclaiming space with 3-22
- scans and asynchronous prefetch 18-4
- segments and 17-12
- select operations and 4-5
- showplan messages about 9-27
- size of 6-5, 6-7, 6-16
- sorts and 7-26
- space requirements 15-17
- structure 4-4
- Clustered table, `sp_sysmon` report
 - on 24-41
- Collapsing tables 2-12
- Columns
 - artificial 7-51
 - average length in object size
 - calculations 6-26
 - datatype sizes and 6-14
 - derived 2-11
 - fixed- and variable-length 6-14
 - image* 6-28 to 6-29
 - redundant in database design 2-11
 - splitting tables 2-16
 - text* 6-28 to 6-29
 - unindexed 3-2
 - values in, and normalization 2-5
- Committed transactions, `sp_sysmon`
 - report on 24-38
- Compiled objects 16-5
 - data cache size and 16-6
- Composite indexes 7-35
 - advantages of 7-37
- compute clause
 - showplan messages for 9-16
- Concurrency
 - deadlocks and 5-28
 - locking and 5-3, 5-28
 - SMP environment 21-23
- Configuration (Server)
 - housekeeper task 21-18
 - I/O 16-13
 - lock limit 5-46
 - memory 16-2
 - named data caches 16-12
 - network packet size 20-3
 - number of rows per page 5-37
 - parallel query processing 13-12
 - performance monitoring and 24-4
 - `sp_sysmon` and 24-3
- Connections
 - client 21-1
 - cursors and 12-17
 - opened (`sp_sysmon` report on) 24-23
 - packet size 20-3
 - sharing 20-9
 - simultaneous in SMP systems 21-13
- Consistency
 - checking databases 6-9
 - data and performance 2-18
 - transactions and 5-2
- Constants `li`
- Constraints
 - primary key 7-6
 - unique 7-6
- Consumer process 15-5, 15-20
- Contention 24-4
 - address locks 24-26
 - avoiding with clustered indexes 4-1
 - data cache 16-27
 - data cache spinlock 24-76
 - device semaphore 24-94
 - disk devices 24-28

- disk I/O 16-39, 17-5, 24-88
- disk structures 24-28
- disk writes 24-25
- I/O device 17-5, 24-28
- inserts 17-16
- last page of heap tables 3-15, 24-63
- lock 24-25, 24-62
- logical devices and 17-2
- log semaphore requests 24-26, 24-47
- `max_rows_per_page` and 5-35
- partitions to avoid 17-14
- reducing 5-33
- SMP servers and 21-23
- spinlock 16-27, 24-76
- system tables in *tempdb* 19-11
- transaction log writes 3-22
- underlying problems 17-3
- yields and 24-24
- context* column of `sp_lock` output 5-39
- Context switches 24-23
- Controller, device 17-5
- Control pages for partitioned tables 17-19
 - updating statistics on 17-30
- Conventions
 - used in manuals xlix
- Conversion
 - datatypes 11-7
 - in lists to or clauses 8-23
 - parameter datatypes 11-8
 - subqueries to equijoins 8-29
 - ticks to milliseconds, formula for 8-8
- Coordinating process 13-3, 15-6
- Correlated subqueries
 - `showplan` messages for 9-52
- Cost
 - parallel clustered index partition scan 14-8
 - parallel hash-based table scan 14-7
 - parallel nonclustered index
 - hash-based scan 14-10
 - parallel partition scan 14-6
 - point query 7-20
 - range query using clustered index 7-20
 - range query using nonclustered index 7-21, 7-23
 - sort operations 7-25
 - `count(*)` aggregate function
 - exists compared to 11-4
 - optimization of 8-27
 - `count` aggregate function
 - optimization of 8-27
 - Counters, internal 24-2
 - Covered queries
 - index covering 3-3
 - specifying cache strategy for 10-13
 - unique indexes and 10-24
 - Covering nonclustered indexes
 - aggregate optimization and 8-26
 - asynchronous prefetch and 18-4
 - configuring I/O size for 16-32
 - cost 7-22
 - disadvantages of 7-38
 - eliminating fragmentation 16-38
 - large I/O for 10-10
 - non-equality operators and 8-10
 - range query cost 7-21
 - rebuilding 7-48
 - `showplan` message for 9-30
- CPU
 - affinity 21-21
 - checkpoint process and usage 24-14
 - guidelines for parallel queries 13-20
 - processes and 24-11
 - saturation 13-19, 13-21
 - Server use while idle 24-12
 - `sp_sysmon` report and 24-9
 - ticks 8-7
 - time 8-7
 - utilization 13-19, 13-23, 21-13
 - yielding and overhead 24-15
 - yields by engine 24-14
 - `cpuaffinity` (dbcc tune parameter) 21-21
 - `cpu grace time` configuration parameter
 - CPU yields and 21-9
 - CPU usage

- applications, `sp_sysmon` report
 - on 24-34
- CPU-intensive queries 13-19
- deadlocks and 5-29
- housekeeper task and 21-17
- logins, `sp_sysmon` report on 24-34
- lowering 24-13
- monitoring 21-19
- `sp_sysmon` report on 24-11
- CPU usages
 - parallel queries and 13-23
- create database command
 - parallel I/O 17-2
- create index command
 - distributing data with 17-24
 - fillfactor option 21-24
 - locks acquired by 5-11, 23-3
 - logging considerations of 15-18
 - `max_rows_per_page` option 21-24
 - number of sort buffers parameter
 - and 15-2, 15-11 to 15-15
 - parallel configuration and 17-24
 - parallel sort and 17-24
 - segments and 23-4
 - `sorted_data` option 23-4
 - space requirements 15-17
 - sub-index creation 15-5
 - with consumers clause and 15-9
- cursor rows option, set 12-16
- Cursors
 - close on endtran option 5-27
 - execute 12-5
 - indexes and 12-6
 - isolation levels and 5-26, 12-13
 - locking and 5-26 to 5-28, 12-4
 - modes 12-6
 - multiple 12-17
 - or strategy optimization and 8-25
 - read-only 12-6
 - shared keyword in 5-27
 - stored procedures and 12-5
 - updatable 12-6
- Cycle, execution 21-6

D

Data

- consistency 2-18, 5-2
- little-used 2-15
- `max_rows_per_page` and storage 5-35
- row size calculation 6-26
- storage 3-1 to 3-23, 17-5
- uniqueness 4-1

Database design 2-1 to 2-18

- collapsing tables 2-12
- column redundancy 2-11
- indexing based on 7-47
- logical keys and index keys 7-31
- normalization 2-3
- ULC flushes and 24-45

Database devices 17-3

- parallel queries and 13-21, 17-5
- sybsecurity* 17-7
- tempdb* 17-6

Database objects

- binding to caches 3-16
- placement 17-1 to 17-45
- placement on segments 17-1
- storage 3-1 to 3-23

Databases

- See also* Database design
- creation speed 23-1
- devices and 17-5
- lock promotion thresholds for 5-46
- placement 17-1

Data caches 16-7 to 16-39, 21-14

- aging in 3-16
- binding objects to 3-16
- cache hit ratio 16-10
- contention 24-76
- data modification and 3-19, 16-9
- deletes on heaps and 3-20
- fetch-and-discard strategy 3-18
- fillfactor and 16-39
- flushing during table scans 7-17
- guidelines for named 16-27
- hot spots bound to 16-12
- inserts to heaps and 3-19
- joins and 3-18

- large I/O and 16-13
- management, `sp_sysmon` report
 - on 24-66
- named 16-12 to 16-35
- page aging in 16-8
- parallel sorting and 15-13, 15-16
- sizing 16-23 to 16-33
- sort buffers and 15-13
- spinlocks on 16-12, 24-76
- strategies chosen by optimizer 16-18
- subquery cache 8-32
- `tempdb` bound to own 16-13, 19-10
- transaction log bound to own 16-13
- updates to heaps and 3-20
- wash marker 3-16
- Data integrity
 - application logic for 2-17
 - denormalization effect on 2-9
 - isolation levels and 5-14
 - managing 2-16
- Data modification
 - data caches and 3-19, 16-9
 - heap tables and 3-12
 - log space and 23-6
 - nonclustered indexes and 7-34
 - number of indexes and 7-4
 - recovery interval and 16-39
 - `showplan` messages 9-9
 - transaction log and 3-22
 - update modes 8-34, 9-9
- Data pages 3-3 to 3-22
 - clustered indexes and 4-4
 - computing number of 6-15
 - fillfactor effect on 6-26
 - full, and insert operations 4-7
 - limiting number of rows on 5-35
 - linking 3-5
 - partially full 3-21
 - prefetching 10-12
 - text and image 3-6
- Datatypes
 - average sizes for 6-26
 - choosing 7-34, 7-51
 - joins and 11-6
 - mismatched 10-19
 - numeric compared to character 7-51
- `dbcc` (Database Consistency Checker)
 - asynchronous prefetch and 18-5
 - configuring asynchronous prefetch
 - for 18-14
 - disadvantages of 6-9
 - isolation levels and 5-14
 - large I/O for 16-14
 - trace flags 10-17
- `dbcc traceon(302)` 10-17 to 10-29
- `dbcc tune`
 - `ascinserts` 24-55
 - `cleanup` 23-10
 - `cpuaffinity` 21-22
 - `deviochar` 24-88
 - `doneinproc` 24-98
 - `maxwritedes` 24-25
- deadlock checking period configuration
 - parameter 5-31
- Deadlocks 5-28 to 5-31, 5-40, 5-42, 5-45
 - avoiding 5-30
 - defined 5-28
 - delaying checking 5-31
 - descending scans and 7-30
 - detection 5-29, 5-40, 5-42, 5-45, 24-64
 - error messages 5-30
 - percentage 24-62
 - performance and 5-32
 - searches 24-65
 - `sp_sysmon` report on 24-62
 - statistics 24-64
 - worker process example 5-43
- `deallocate cursor` command
 - memory and 12-5
- Debugging aids
 - `dbcc traceon(302)` 10-17
 - `set forceplan on` 10-3
- Decision support system (DSS)
 - applications
 - execution preference 22-21
 - named data caches for 16-13
 - network packet size for 20-3
 - parallel queries and 13-2, 13-23

- declare cursor command
 - memory and 12-5
- Default settings
 - auditing 16-41
 - audit queue size 16-42
 - index fillfactor 7-7
 - index statistics 8-19
 - max_rows_per_page 5-36
 - network packet size 20-3
 - number of tables optimized 10-8
 - optimizer 10-27
- Deferred delete statistics 24-42
- Deferred index updates 8-39
- Deferred updates 8-39
 - scan counts and 7-14
 - showplan messages for 9-10
 - statistics 24-42
- Degree of parallelism 13-11, 14-13 to 14-16
 - definition of 14-13
 - joins and 14-18, 14-20
 - optimization of 14-14
 - parallel sorting and 15-9
 - query-level 13-15
 - run-time adjustment of 14-16, 14-30 to 14-36
 - server-level 13-12
 - session-level 13-14
 - upper limit to 14-14
- delete command
 - locks created by 5-11
 - transaction isolation levels and 5-16
- Delete operations
 - clustered indexes 4-10
 - heap tables 3-14
 - index maintenance and 24-51
 - nonclustered indexes 4-20
 - object size and 6-3
 - statistics 24-42
- Demand locks 5-7
 - sp_lock report on 5-39
- Denormalization 2-8
 - application design and 2-17
 - batch reconciliation and 2-18
- derived columns 2-11
- disadvantages of 2-9
- duplicating tables and 2-13
 - management after 2-16
 - performance benefits of 2-9
 - processing costs and 2-9
 - redundant columns 2-11
 - techniques for 2-10
 - temporary tables and 19-5
- Density
 - index, and joins 8-18
 - table row 3-4
- Density table 7-44
 - joins and 10-29
 - optimizer and 7-46
- Derived columns 2-11
- Descending order (desc keyword) 7-27, 7-29
 - covered queries and 7-29
- Descending scans
 - deadlocks and 7-30
- Descending scan showplan message 9-28
- Detecting deadlocks 5-40, 5-42, 5-45
- Devices
 - activity detail 24-93
 - adding 24-4
 - adding for partitioned tables 17-37, 17-41
 - object placement on 17-1
 - partitioned tables and 17-41
 - RAID 13-21, 17-19
 - semaphores 24-94
 - throughput, measuring 17-19
 - using separate 21-24
- deviochar (dbcc tune parameter) 24-88
- Direct updates 8-34
 - cheap 8-36
 - expensive 8-37
 - in-place 8-35
 - statistics 24-42
- Dirty pages
 - checkpoint process and 16-8
 - wash area and 16-8
- Dirty reads 5-3

- modify conflicts and 24-28
 - preventing 5-14
 - requests 24-83
 - restarts 24-75
 - sp_sysmon report on 24-75
 - transaction isolation levels and 5-12
 - Discarded (MRU) buffers, sp_sysmon
 - report on 24-71, 24-81
 - Disk devices
 - adding 24-4
 - average I/Os 24-17
 - contention 24-28
 - I/O management report (sp_sysmon) 24-88
 - I/O report (sp_sysmon) 24-16
 - I/O speed 13-20
 - I/O structures 24-91
 - parallel queries and 13-17, 13-19
 - parallel sorting and 15-16, 15-17
 - performance and 17-1 to 17-45
 - transaction log and
 - performance 24-27
 - write operations 24-25
 - Disk I/O 21-14
 - application statistics 24-35
 - performing 21-16
 - queue 21-14
 - sp_sysmon report on 24-88
 - disk i/o structures configuration
 - parameter 24-91
 - asynchronous prefetch and 18-7
 - Disk mirroring
 - device placement 17-9
 - performance and 17-2
 - distinct keyword
 - parallel optimization of 15-1
 - showplan messages for 9-19, 9-57
 - worktables for 7-15
 - Distribution map 15-4, 15-20
 - Distribution pages 7-42
 - matching values on 10-25
 - nonexistent 10-27
 - optimizer and 10-24
 - space calculations and 6-26
 - sysindexes storage 7-50
 - Distribution table 7-42
 - missing 10-27
 - optimizer and 7-46
 - Dropping
 - indexes specified with index 10-10
 - dump database command
 - parallel sorting and 15-3, 15-18
 - Duplicate rows
 - removing from worktables 8-24
 - Duplication
 - tables 2-13
 - update performance effect of 8-39
 - Dynamic indexes 8-24
 - showplan message for 9-33
- ## E
- EC (execution class) 22-2
 - attributes 22-3
 - End transaction, ULC flushes and 24-45
 - Engine affinity, task 22-3, 22-5
 - example 22-8
 - Engines 21-1
 - busy 24-12
 - “config limit” 24-91
 - connections and 24-23
 - CPU affinity 21-21
 - CPU report and 24-13
 - defined 21-2
 - functions and scheduling 21-11 to 21-13
 - monitoring performance 24-4
 - network 21-12
 - number of 13-19
 - outstanding I/O 24-91
 - scheduling 21-11
 - utilization 24-12
 - Environment analysis 22-28
 - Equijoins
 - subqueries converted to 8-29
 - Equivalentents in search arguments 8-11
 - Error logs
 - procedure cache size in 16-5

- Error messages
 - deadlocks 5-30
 - procedure cache 16-7
 - process_limit_action 14-32
 - run-time adjustments 14-32
 - Errors
 - packet 20-6
 - procedure cache 16-4
 - Escalation, lock 5-47
 - Estimated cost 8-5
 - fast and slow query processing 8-4
 - I/O, reported by showplan 9-38
 - indexes 8-3
 - joins 8-14
 - materialization 8-31
 - or clause 8-24
 - reformatting 8-17
 - search arguments 8-9
 - subquery optimization 8-34
 - Exclusive locks
 - intent deadlocks 24-64
 - page 5-5
 - page deadlocks 24-64
 - sp_lock report on 5-39
 - table 5-6
 - table deadlocks 24-64
 - Execute cursors
 - memory use of 12-5
 - Execution 21-16
 - attributes 22-1
 - cycle 21-6
 - mixed workload precedence 22-21
 - precedence and users 22-22
 - preventing with set noexec on 9-1
 - ranking applications for 22-1 to 22-29
 - stored procedure precedence 22-22
 - system procedures for 22-7
 - time statistics from set statistics time on 8-7
 - Execution class 22-2
 - attributes 22-3
 - predefined 22-2
 - user-defined 22-3
 - Execution objects 22-2
 - behavior 22-27
 - performance hierarchy 22-1 to 22-29
 - scope 22-13
 - Execution precedence
 - among applications 22-7
 - assigning 22-2
 - scheduling and 22-10
 - Existence joins
 - parallel considerations for 14-21
 - showplan messages for 9-58
 - subqueries flattened to 8-28
 - exists keyword
 - compared to count 11-4
 - optimizing searches 11-2
 - parallel joins and 14-21
 - parallel optimization of 14-28
 - showplan messages for 9-58
 - subquery optimization and 8-28
 - Expensive direct updates 8-37, 8-38
 - Expressions
 - types of li
 - in where clauses 10-27
 - Expression subqueries
 - optimization of 8-29
 - showplan messages for 9-55
 - Extended stored procedures
 - sp_sysmon report on 24-36
 - Extents
 - dbcc reports on 6-8
 - partitioned tables and extent stealing 17-29
 - space allocation and 3-5
- F**
- FALSE, return value of 8-28
 - Family of worker processes 13-3
 - Fetch-and-discard cache strategy 3-18
 - Fetching cursors
 - locking and 5-28
 - memory and 12-5
 - Fillfactor
 - advantages of 7-52
 - data cache performance and 16-39

- disadvantages of 7-53
 - index creation and 7-33, 7-51
 - index page size and 6-25
 - locking and 5-35
 - max_rows_per_page compared to 5-36
 - page splits and 7-52
 - SMP environment 21-24
 - First normal form 2-4
 - See also* Normalization
 - First page
 - allocation page 3-8
 - partitions 17-19
 - text pointer 3-6
 - Fixed-length columns
 - calculating space for 6-11
 - data row size of 6-14
 - for index keys 7-34
 - indexes and update modes 8-44
 - index row size and 6-15
 - null values in 11-7
 - overhead 7-34
 - Flattened subqueries 8-28, 14-27
 - showplan messages for 9-47
 - Floating-point data *li*
 - forceplan option, set 10-3
 - alternatives 10-7
 - risks of 10-7
 - Foreign keys
 - denormalization and 2-9
 - for load option
 - performance and 23-1
 - Formulas
 - cache hit ratio 16-11
 - factors affecting 6-25 to 6-29
 - index distribution steps 7-43
 - nonclustered hash-based scan
 - cost 14-10
 - optimizer 7-46
 - parallel join cost 14-17
 - serial join cost 14-17
 - table or index sizes 6-11 to 6-29
 - tempdb size 19-6
 - for update option, declare cursor
 - optimizing and 12-15
 - Fragmentation, data 16-37
 - effects on asynchronous prefetch 18-8
 - large I/O and 24-72
 - page chain 18-8
 - Free checkpoints 24-87
 - Free writes 21-17
 - from keyword
 - order of tables in clause 8-20
 - Full ULC, log flushes and 24-45
- ## G
- Global Allocation Map (GAM)
 - pages 3-8
 - goto keyword
 - optimizing queries and 11-2
 - Grabbed dirty, sp_sysmon report on 24-79
 - group by clause
 - showplan messages for 9-13, 9-15
 - Group commit sleeps, sp_sysmon report
 - on 24-27
- ## H
- Hardware 1-6
 - network 20-8
 - parallel query processing
 - guidelines 13-20
 - ports 20-13
 - terminology 17-3
 - Hash-based scans
 - asynchronous prefetch and 18-13
 - heap tables and 14-12
 - I/O and 14-4
 - indexing and 14-12
 - joins and 17-5
 - limiting with set
 - scan_parallel_degree 13-14
 - nonclustered indexes and 14-9 to 14-10, 14-12
 - table scans 14-6 to 14-7
 - worker processes and 14-4
 - Header information
 - data pages 3-3

- index pages 4-4
- packet 20-3
- “proc headers” 16-6
- Heading, `sp_sysmon` report 24-11
- Heap tables 3-11 to 3-23
 - `bcp` (bulk copy utility) and 23-9
 - delete operations 3-14
 - deletes and pages in cache 3-20
 - guidelines for using 3-21
 - I/O and 3-15
 - I/O inefficiency and 3-21
 - insert operations on 3-12
 - inserts 24-41
 - inserts and pages in cache 3-19
 - lock contention 24-63
 - locking 3-13
 - maintaining 3-21
 - performance limits 3-13
 - select operations on 3-12, 3-19
 - updates and pages in cache 3-20
 - updates on 3-14
- High priority users 22-22
- Historical data 2-15
- `holdlock` keyword
 - locking 5-21
 - `shared` keyword and 5-27
- Horizontal table splitting 2-14
- Hot spots 22-22
 - avoiding 5-34
 - binding caches to 16-12
- housekeeper free write percent configuration
 - parameter 21-18, 24-87
- Housekeeper task 21-17 to 21-19
 - batch write limit 24-87
 - checkpoints and 24-86
 - recovery time and 16-40
 - `sp_sysmon` and 24-85
- I**
- I/O
 - See also* Large I/O
 - access problems and 17-3
 - asynchronous prefetch 18-1 to 18-16
 - balancing load with segments 17-13
 - batch limit 24-25
 - `bcp` (bulk copy utility) and 23-9
 - checking 24-16
 - completed 24-92
 - CPU and 21-19, 24-13
 - `create database` and 23-2
 - `dbcc` commands and 6-6
 - default caches and 3-16
 - delays 24-91
 - device contention and 24-28
 - devices and 17-2
 - direct updates and 8-34
 - disk 21-16
 - efficiency on heap tables 3-21
 - estimating 7-17
 - heap tables and 3-15
 - increasing size of 3-15
 - limits 24-90
 - limits, effect on asynchronous prefetch 24-73
 - maximum outstanding 24-90
 - memory and 16-1
 - memory pools and 16-12
 - named caches and 16-12
 - network 21-13, 21-17
 - optimizer estimates of 10-19
 - pacing 24-25
 - parallel for `create database` 17-2
 - performance and 17-4
 - physical compared to logical 10-28
 - `prefetch` keyword 10-11
 - range queries and 10-10
 - recovery interval and 23-6
 - requested 24-92
 - saturation 13-19
 - saving with reformatting 8-17
 - select operations on heap tables and 3-19
 - server-wide and database 17-6, 24-88
 - `showplan` messages for 9-37
 - `sp_spaceused` and 6-6
 - specifying size in queries 10-10
 - spreading between caches 19-10

- statistics information 7-9
- structures 24-91
- total 24-93
- total estimated cost **showplan**
 - message 9-38
- transaction log and 3-22
- update operations and 8-37
- i/o polling process count configuration**
 - parameter
 - network checks and 24-16
- IDENTITY columns**
 - cursors and 12-7
 - indexing and performance 7-32
- Idle CPU, **sp_sysmon** report on 24-14
- if...else conditions**
 - optimizing 11-2
- image datatype**
 - large I/O for 16-14
 - page number estimation 6-28 to 6-29
 - page size for storage 3-7
 - storage on separate device 3-6, 17-13
- indexalloc option, dbcc**
 - index size report 6-7
- Index covering**
 - definition 3-3
 - showplan** messages for 9-30
 - sort operations and 7-29
- Index descriptors, sp_sysmon** report
 - on 24-59
- Indexes 4-2 to 4-26**
 - access through 3-3, 4-1
 - add levels statistics 24-56
 - avoiding sorts with 7-25
 - B-trees 4-4
 - bulk copy and 23-7
 - cache replacement policy for 16-34
 - choosing 3-3
 - computing number of pages 6-16
 - costing 10-23
 - creating 15-1, 23-3
 - cursors using 12-6
 - denormalization and 2-9
 - design considerations 7-1
 - dropping infrequently used 7-47
 - dynamic 8-24
 - fillfactor and 7-51
 - forced 10-22
 - guidelines for 7-33
 - information about 7-49
 - intermediate level 4-3
 - leaf level 4-3
 - leaf pages 4-14
 - locking with 5-5
 - maintenance statistics 24-50
 - management 24-49
 - max_rows_per_page** and 5-36
 - multiple 21-23
 - number allowed 7-6
 - page chains 4-4
 - parallel creation of 15-1
 - performance and 4-1 to 8-42
 - performance management tools 7-7, 7-8
 - rebuilding 7-48
 - recovery and creation 23-4
 - root level 4-3
 - selectivity 7-3
 - size estimation of 6-1 to 6-29
 - size of 6-1
 - size of entries and performance 7-4
 - SMP environment and multiple 21-23
 - sort order changes 7-48
 - sp_spaceused** size report 6-5
 - specifying for queries 10-8
 - temporary tables and 19-3, 19-13
 - types of 4-2
 - update modes and 8-43
 - update operations and 8-35, 8-36
 - usefulness of 3-12
- Index keys, logical keys and 7-31**
- Index pages**
 - fillfactor effect on 6-25, 7-53
 - limiting number of rows on 5-35
 - page splits for 4-9
 - storage on 4-2
- Information (Server)**
 - CPU usage 21-19
 - I/O statistics 7-9

- indexes 7-49
- storage 3-3
- Initializing
 - text or image pages 6-28
- in keyword
 - matching index scans and 9-32
 - optimization of 8-23
 - subquery optimization and 8-28
- Inner tables of joins 8-16
- In-place updates 8-35
- insert command
 - contention and 5-34
 - locks created by 5-11
 - transaction isolation levels and 5-16
- Insert operations
 - clustered indexes 4-7
 - clustered table statistics 24-41
 - contention 17-16
 - heap tables and 3-12
 - heap table statistics 24-41
 - index maintenance and 24-50
 - logging and 19-11
 - nonclustered indexes 4-19
 - page split exceptions and 4-9
 - partitions and 17-14
 - performance of 17-2
 - rebuilding indexes after many 7-48
 - statistics 24-40
 - total row statistics 24-41
- Integer data
 - in SQL `li`
 - optimizing queries on 10-19, 11-1
- Intent table locks 5-6
 - deadlocks and 5-31
 - `sp_lock` report on 5-39
- Intermediate levels of indexes 4-3
- Isolation levels **5-12 to 5-26**
 - cursors 5-26, 12-13
 - default 5-20
 - dirty reads 5-14
 - nonrepeatable reads 5-15
 - phantoms 5-16
 - transactions 5-12

J

Joins

- choosing indexes for 7-32
- data cache and 3-18
- datatype compatibility in 7-34, 11-7
- denormalization and 2-8
- density table 10-29
- derived columns instead of 2-11
- distribution pages and 10-29
- evaluating 10-21
- existence 8-28, 14-21
- hash-based scan and 17-5
- index density 8-18
- indexing by optimizer 8-14
- many tables in 8-20, 8-21
- normalization and 2-4
- number of tables considered by
 - optimizer 10-8
- optimizing 8-13, 10-18
- optimizing parallel 14-17 to 14-21
- or clause optimization 11-5
- parallel optimization of 14-17 to
 - 14-21, 14-23 to 14-26
- process of 8-14
- scan counts for 7-13
- selectivity and optimizer 10-28
- table order in 8-20, 10-3
- table order in parallel 14-17 to 14-21,
 - 14-23 to 14-26
- temporary tables for 19-5
- union operator optimization 11-5
- updates using 8-35, 8-39

K

Kernel

- engine busy utilization 24-12
- utilization 24-11

Keys, index

- choosing columns for 7-32
- clustered and nonclustered indexes
 - and 4-2
- composite 7-35
- density and 8-18

- density table of 7-44
- distribution table 7-42
- logical keys and 7-31
- monotonically increasing 4-9
- showplan messages for 9-30
- size 7-6
- size and performance 7-34
- unique 7-33
- update operations on 8-35
- Key values
 - distribution table for 7-42
 - index statistics and 10-24
 - index storage 4-2
 - order for clustered indexes 4-4
 - overflow pages and 4-10
- L**
- Large I/O
 - asynchronous prefetch and 18-11
 - denied 24-72, 24-82
 - effectiveness 24-72
 - fragmentation and 24-72
 - named data caches and 16-13
 - pages cached 24-72, 24-83
 - pages used 24-72, 24-83
 - performed 24-71, 24-82
 - pool detail 24-83
 - restrictions 24-82
 - total requests 24-72, 24-82
 - usage 24-71, 24-81
- Last log page writes in `sp_sysmon`
 - report 24-27
- Last page locks on heaps in `sp_sysmon`
 - report 24-63
- Leaf levels of indexes 4-3
 - fillfactor and number of rows 6-26
 - large I/O and 16-13
 - queries on 3-2
 - row size calculation 6-20
- Leaf pages 4-14
 - calculating number in index 6-21
 - limiting number of rows on 5-35
- Levels
 - indexes 4-3
 - locking 5-37
 - tuning 1-3 to 1-7
- Lightweight process 21-3
- like keyword
 - optimization 8-11
- Limits
 - parallel query processing 13-11, 13-14
 - parallel sort 13-11
 - worker processes 13-11, 13-14
- Listeners, network 20-12
- Load balancing for partitioned
 - tables 17-28
 - maintaining 17-44
- Local backups 23-6
- Local variables
 - optimizer and 10-27
- Lock detection 5-45
- Locking **5-1 to 5-47**
 - concurrency 5-3
 - contention, reducing **5-32 to 5-38**
 - contention and 24-25
 - control over 5-3, 5-4
 - create index and 23-3
 - cursors and 5-26
 - by `dbcc` commands 6-6
 - deadlocks 5-28 to 5-31
 - entire table 5-4
 - example of 5-23 to 5-26
 - forcing a write 5-7
 - for update clause 5-26
 - heap tables and inserts 3-13
 - `holdlock` keyword 5-20
 - indexes used 5-5
 - isolation levels and 5-12 to 5-26
 - last page inserts and 7-31
 - `noholdlock` keyword 5-20, 5-23
 - or strategy optimization and 8-25
 - overhead 5-3
 - page and table, controlling 5-12, 5-47
 - performance 5-32
 - read committed clause 5-21
 - read uncommitted clause 5-21, 5-23
 - reducing contention 5-33

- row density and 3-4
- serializable clause 5-21
- shared keyword 5-20, 5-23
- sp_lock report on 5-38
- sp_sysmon report on 24-62
- tempdb and 19-10
- transactions and 5-3
- worktables and 19-11
- lock promotion HWM configuration
 - parameter 5-48
- lock promotion LWM configuration
 - parameter 5-48
- lock promotion PCT configuration
 - parameter 5-49
- Lock promotion thresholds **5-46 to 5-51**
 - database 5-50
 - default 5-50
 - dropping 5-51
 - precedence 5-50
 - promotion logic 5-49
 - server-wide 5-50
 - sp_sysmon report on 24-66
 - table 5-50
- Locks
 - address 24-26
 - blocking 5-38
 - deadlock percentage 24-62
 - demand 5-7
 - escalation 5-47
 - exclusive page 5-5
 - exclusive table 5-6
 - granularity 5-3
 - intent table 5-6
 - limits 5-11
 - “lock sleep” status 5-38
 - page 5-4
 - parallel queries and 5-43
 - reporting on 5-38
 - shared page 5-4
 - shared table 5-6
 - size of 5-3
 - “sleeping” status 5-44
 - sp_sysmon report on 24-62
 - summary of 5-23
 - “sync sleep” status 5-44
 - table 5-6
 - table vs. page 5-47
 - total requests 24-62
 - types of 5-3, 5-39
 - update page 5-5
 - viewing 5-38
 - worker processes and 5-9
- locktype column of sp_lock output 5-39
- Logging
 - bulk copy and 23-7
 - minimizing in tempdb 19-11
 - parallel sorting and 15-18
- Log I/O size
 - group commit sleeps and 24-27
 - matching 16-16
 - tuning 16-26, 24-27
 - using large 16-30
- Logical database design 2-1
- Logical device name 17-3
- Logical expressions li
- Logical keys, index keys and 7-31
- Logical Process Manager 22-1
- Login process 20-9, 21-16
- Logins 21-12
- Log scan showplan message 9-37
- Log semaphore requests 24-47
- Look-ahead set **18-2**
 - dbcc and 18-5
 - during recovery 18-3
 - nonclustered indexes and 18-4
 - sequential scans and 18-4
- Lookup tables, cache replacement policy
 - for 16-34
- Loops
 - runnable process search count and 24-13, 24-14
 - showplan messages for nested iterations 9-8
- Lower bound of range query 10-26
- LRU replacement strategy 3-16
 - buffer grab in sp_sysmon report 24-78
 - I/O and 7-16, 7-17
 - showplan messages for 9-38

specifying 10-14

M

Magic numbers 10-27

Maintenance tasks 23-1 to 23-10

forced indexes 10-10

forceplan checking 10-3

indexes and 24-50

performance and 17-2

Managing denormalized data 2-16

Map, Object Allocation. *See* Object Allocation Map (OAM)

Matching index scans 4-21

showplan message 9-32

Materialized subqueries 8-29

showplan messages for 9-48

max_rows_per_page option

fillfactor compared to 5-36

locking and 5-35

select into effects 5-36

max aggregate function

min used with 8-27

optimization of 8-27, 11-5

max async i/os per engine configuration parameter

asynchronous prefetch and 18-7

tuning 24-91

max async i/os per server configuration parameter

asynchronous prefetch and 18-7

tuning 24-91

Maximum outstanding I/Os 24-90

Maximum ULC size, sp_sysmon report on 24-46

max parallel degree configuration parameter 13-12, 14-15

sorts and 15-7

max scan parallel degree configuration parameter 13-12, 14-14

maxwritedes (dbcc tune parameter) 24-25

Memory

allocated 24-85

configuration parameters for 16-2

cursors and 12-4

I/O and 16-1

named data caches and 16-12

network packets and 20-4

performance and 16-1 to 16-42

released 24-85

shared 21-10

sp_sysmon report on 24-85

Merge runs, parallel sorting 15-5, 15-11

reducing 15-12

Merging index results 15-6

Messages

See also Errors

dbcc traceon(302) 10-17 to 10-29

deadlock victim 5-29

dropped index 10-10

range queries 10-26

showplan 9-1 to 9-59

turning off TDS 24-98

min aggregate function

max used with 8-27

optimization of 8-27, 11-5

Mixed workload execution

priorities 22-21

Model, SMP process 21-10 to 21-13

Modes of disk mirroring 17-10

“Modify conflicts” in sp_sysmon report 24-28

Monitoring

CPU usage 21-19

data cache performance 16-10

indexes 7-7

index usage 7-47

network activity 20-6

performance 1-3, 24-2

MRU replacement strategy 3-16

asynchronous prefetch and 18-12

disabling 10-14

I/O and 7-17

showplan messages for 9-38

specifying 10-14

Multicolumn index. *See* Composite indexes

Multidatabase transactions 24-40, 24-45

- Multiple network engines 21-12
- Multiple network listeners 20-12
- Multiprocessing architecture, Adaptive Server SMP 21-14 to 21-17
- Multitasking 21-5
- Multithreading 21-1

N

Names

- column, in search arguments 8-10
- index, in `showplan` messages 9-27
- index clause and 10-9
- index prefetch and 10-12
- prefetch and 10-12

- Nested iterations 8-13

Nesting

- `showplan` messages for 9-52
- temporary tables and 19-14

- Network engines 21-12

- manager 21-16

- Network I/O 21-13, 21-14

- application statistics 24-35
 - performing 21-17

Networks

- blocking checks 24-15
- bottlenecks 21-13
- controllers 21-13
- cursor activity of 12-10
- delayed I/O 24-97
- hardware for 20-8
- I/O management 24-94
- `i/o polling process count` and 24-16
- multiple listeners 20-12
- packet size 24-29
- packets received 24-28
- packets sent 24-29
- performance and 20-1 to 20-13
- ports 20-13
- queues 21-14
- reducing traffic on 20-7, 23-10, 24-98
- `sp_sysmon` report on 24-14
- total I/O checks 24-15
- tuning issues 1-6

- `noholdlock` keyword, `select` 5-23

- Non-blocking network checks, `sp_sysmon` report on 24-15

- Nonclustered indexes 4-2

- asynchronous prefetch and 18-4
 - covered queries and sorting 7-29

- `create index` requirements 15-8

- `dbcc indexalloc` report on 6-8

- definition of 4-13

- delete operations 4-20

- estimating size of 6-20 to 6-22

- guidelines for 7-33

- hash-based scans 14-9 to 14-10

- insert operations 4-19

- maintenance report 24-50

- number allowed 7-6

- offset table 4-14

- point query cost 7-20

- range query cost 7-21, 7-23

- row IDs 4-14

- select operations 4-17

- size of 4-14, 6-5, 6-7, 6-20

- sorting and 7-28

- structure 4-16

- Non-leaf rows 6-21

- Nonmatching index scans **4-22 to 4-23**

- non-equality operators and 8-10

- Nonrepeatable reads 5-15

- Normalization 2-3

- first normal form 2-4

- joins and 2-4

- second normal form 2-5

- temporary tables and 19-5

- third normal form 2-6

- `not exists` keyword

- optimizing searches 11-2

- `not in` keyword

- optimizing searches using 11-2

- Null columns

- optimizing updates on 8-43

- storage of rows 3-3

- storage size 6-14

- `text` and `image` 6-28

- variable-length 7-34

- Null values
 - datatypes allowing 7-34, 11-7
 - text* and *image* columns 6-28
 - variable-length columns and 11-7
- Number (quantity of)
 - bytes per index key 7-6
 - checkpoints 24-86
 - clustered indexes 4-2
 - cursor rows 12-16
 - engines 13-19
 - indexes 21-23
 - indexes per table 7-6
 - locks in the system 5-46
 - locks on a table 5-48
 - nonclustered indexes 4-2
 - OAM pages 6-22
 - packet errors 20-6
 - packets 20-4
 - procedure (“proc”) buffers 16-5
 - processes 21-4
 - rows (rowtotal), estimated 6-4
 - rows, optimizer and 10-23
 - rows on a page 5-35
 - tables considered by optimizer 10-8
- number of sort buffers configuration
 - parameter
 - parallel sorting and 15-2, 15-11 to 15-15
 - parallel sort messages and 15-19
- number of worker processes configuration
 - parameter 13-12
- Numbers
 - magic 10-27
 - row offset 4-13
 - showplan output 9-3
- Numeric expressions li
- O**
- Object Allocation Map (OAM)
 - pages 3-9
 - dbcc commands using 6-6
 - LRU strategy in data cache 3-18
 - object size information in 6-3
 - overhead calculation and 6-17
 - pointers in *sysindexes* 7-50
 - sp_spaceused and 6-3
 - system functions and 6-4
- Object ID on data pages 3-3
- Observing deadlocks 5-40, 5-42, 5-45
- Offset table 4-14
 - nonclustered index selects and 4-17
 - row IDs and 4-13
 - size of 3-4
- Online backups 23-6
- Online transaction processing (OLTP)
 - execution preference
 - assignments 22-21
 - named data caches for 16-13
 - network packet size for 20-3
 - parallel queries and 14-3
- open command
 - memory and 12-5
- Operating systems
 - monitoring Server CPU usage 24-12
 - outstanding I/O limit 24-91
- Operators
 - non-equality, in search
 - arguments 8-10
 - in search arguments 8-10
- Optimization
 - See also* Parallel query optimization
 - cursors 12-5
 - dbcc traceon(302) and 10-17
 - in keyword and 8-23
 - parallel query 14-1 to 14-39
 - subquery processing order 8-34
- Optimizer 8-2 to 8-45, 14-1 to 14-39
 - See also* Parallel query optimization
 - aggregates and 8-26, 14-29
 - cache strategies and 16-18
 - diagnosing problems of 8-1, 14-37
 - dropping indexes not used by 7-47
 - expression subqueries 8-29
 - I/O estimates 10-19
 - indexes and 7-2
 - index statistics and 7-45
 - join order 8-20, 14-17 to 14-21

- join selectivity 10-28
 - nonunique entries and 7-3
 - or clauses and 8-23
 - overriding 10-1
 - page and row count accuracy 10-23
 - parallel queries and 14-1 to 14-39
 - procedure parameters and 11-3
 - quantified predicate subqueries 8-28
 - query plan output 9-2
 - reformatting strategy 8-17, 9-35
 - sources of problems 8-2
 - subqueries and 8-28
 - subquery short-circuiting 8-31
 - temporary tables and 19-12
 - understanding 10-17
 - updates and 8-42
 - viewing with trace flag 302 10-17
 - Order
 - composite indexes and 7-37
 - data and index storage 4-2
 - index key values 4-4
 - joins 8-20, 14-17 to 14-21
 - presorted data and index
 - creation 23-4
 - recovery of databases 23-7
 - result sets and performance 3-21
 - subquery clauses 8-32
 - tables in a join 8-21, 10-3
 - tables in a query 8-20
 - tables in `showplan` messages 9-5
 - order by clause
 - indexes and 4-1
 - parallel optimization of 14-28, 15-1
 - `showplan` messages for 9-19
 - worktables for 7-15, 9-21
 - or keyword
 - estimated cost 8-24
 - matching index scans and 9-32
 - optimization and 8-23
 - optimization of join clauses
 - using 11-5
 - processing 8-23
 - scan counts and 7-13
 - subqueries containing 8-32
 - OR strategy 8-24
 - cursors and 12-14
 - locking and 8-25
 - `showplan` messages for 9-29, 9-33
 - Outer joins 8-15, 14-21
 - Output
 - `dbcc` 6-6
 - optimizer 10-19
 - `showplan` 9-1 to 9-59
 - `sp_estspace` 7-4
 - `sp_spaceused` 6-4
 - Overflow pages 4-10
 - key values and 4-10
 - Overhead
 - calculation (space allocation) 6-22
 - clustered indexes and 3-21
 - CPU yields and 24-15
 - cursors 12-10
 - datatypes and 7-51
 - datatypes and performance 7-34
 - deferred updates 8-39
 - maximum row size and 6-12
 - network packets and 20-4, 24-98
 - nonclustered indexes 7-34
 - object size calculations 6-11
 - parallel query 14-3 to 14-4
 - pool configuration 16-34
 - row and page 6-11
 - single process 21-3
 - `sp_sysmon` 24-2
 - space allocation calculation 6-17
 - variable-length and null
 - columns 6-14
 - variable-length columns 7-34
- P**
- `@@pack_received` global variable 20-6
 - `@@pack_sent` global variable 20-6
 - `@@packet_errors` global variable 20-6
 - Packets, network 20-3
 - average size received 24-97
 - average size sent 24-97
 - received 24-97

- sent 24-97
- size, configuring 20-3, 24-29
- Page allocation to transaction log 24-48
- Page chain kinks
 - asynchronous prefetch and 18-8, 18-15
 - clustered indexes and 18-15
 - defined 18-8
 - heap tables and 18-15
 - nonclustered indexes and 18-15
- Page chains
 - data and index 3-5
 - index 4-4
 - overflow pages and 4-10
 - partitions 17-14
 - placement 17-1
 - text* or *image* data 6-29
 - unpartitioning 17-23
- Page locks 5-3
 - sp_lock* report on 5-39
 - table locks vs. 5-47
 - types of 5-4
- Page requests, *sp_sysmon* report on 24-75
- Pages, control **17-19**
 - syspartitions* and 17-19
 - updating statistics on 17-30
- Pages, data 3-3 to 3-22
 - bulk copy and allocations 23-7
 - calculating number of 6-15
 - fillfactor effect on 6-26
 - fillfactor for SMP systems 21-24
 - linking 3-5
 - max_rows_per_page* and SMP systems 21-24
 - prefetch and 10-12
 - size 3-3
 - splitting 4-7
- Pages, distribution 7-42
 - matching values on 10-25
 - nonexistent 10-27
 - space calculations and 6-26
 - sysindexes* storage 7-50
- Pages, Global Allocation Map (GAM) 3-8
- Pages, index
 - aging in data cache 16-8
 - calculating number of 6-16
 - calculating number of non-leaf 6-21
 - fillfactor effect on 6-25, 7-53
 - fillfactor for SMP systems 21-24
 - leaf level 4-14
 - max_rows_per_page* and SMP systems 21-24
 - shrinks, *sp_sysmon* report on 24-56
 - storage on 4-2
- Pages, OAM (Object Allocation Map) 3-9
 - aging in data cache 16-8
 - number of 6-17, 6-22
 - object size information in 6-3
- Pages, overflow 4-10
- Page splits 24-51
 - avoiding 24-52
 - data pages 4-7
 - disk write contention and 24-25
 - fillfactor effect on 7-52
 - fragmentation and 16-39
 - index maintenance and 24-52
 - index pages and 4-9
 - max_rows_per_page* setting and 5-35
 - nonclustered indexes, effect on 4-8
 - object size and 6-3
 - performance impact of 4-9
 - reducing 7-52
 - retries and 24-55
- page utilization percent configuration
 - parameter
 - object size estimation and 6-12
- Parallel clustered index partition
 - scan 14-7 to 14-9
 - cost of using 14-8
 - definition of 14-7
 - requirements for using 14-8
 - summary of 14-12
- Parallel hash-based table scan 14-6 to 14-7
 - cost of using 14-7
 - definition of 14-6

- requirements for using 14-7
- summary of 14-12
- parallel keyword, select command 14-36
- Parallel nonclustered index hash-based scan 14-9 to 14-10
 - cost of using 14-10
 - summary of 14-12
- Parallel partition scan 14-5 to 14-6
 - cost of using 14-6
 - definition of 14-5
 - example of 14-22
 - requirements for using 14-5
 - summary of 14-12
- Parallel queries
 - locks and 5-43
 - worktables and 14-28
- Parallel query optimization 14-1 to 14-39
 - aggregate queries 14-29
 - definition of 14-1
 - degree of parallelism 14-13 to 14-16
 - examples of 14-21 to 14-36
 - exists clause 14-28
 - join order 14-17 to 14-21, 14-23 to 14-26
 - order by clause 14-28
 - overhead 14-2, 14-3 to 14-4
 - partitioning considerations 14-4
 - requirements for 14-3
 - resource limits 14-39
 - select into queries 14-29 to 14-30
 - serial optimization compared to 14-2
 - single-table scans 14-21 to 14-23
 - speed as goal 14-2
 - subqueries 14-27 to 14-28
 - system tables and 14-3
 - troubleshooting 14-37
 - union operator 14-28
- Parallel query processing **13-2 to 13-28**, 14-1 to 14-39
 - asynchronous prefetch and 18-12
 - configuring for 13-12
 - configuring worker processes 13-14
 - CPU usage and 13-19, 13-20, 13-23
 - deadlock detection 5-42
 - demand locks and 5-9
 - disk devices and 13-19
 - execution phases 13-4
 - hardware guidelines 13-20
 - I/O and 13-19
 - joins and 13-9
 - merge types 13-5
 - object placement and 17-2
 - performance of 17-2
 - query types and 13-2
 - resources 13-18
 - worker process limits 13-12
- Parallel sorting **15-1 to 15-24**
 - clustered index requirements 15-8
 - commands affected by 15-1
 - conditions for performing 15-2
 - configuring worker processes 13-14
 - coordinating process and 15-6
 - degree of parallelism of 15-9, 15-19
 - distribution map 15-4, 15-20
 - dynamic range partitioning for 15-5
 - examples of 15-20 to 15-22
 - logging of 15-18
 - merge runs 15-5
 - merging results 15-6
 - nonclustered index
 - requirements 15-8
 - number of sort buffers parameter and 15-2
 - observation of 15-18 to 15-22
 - overview of 15-3
 - producer process and 15-5
 - range sorting and 15-5
 - recovery and 15-3, 15-18
 - resources required for 15-2, 15-6
 - sampling data for 15-4, 15-20
 - select into/bulk copy/pllsort option and 15-2
 - sort_resources option 15-19
 - sort buffers and 15-11 to 15-12, 15-19
 - sub-indexes and 15-5
 - target segment 15-7
 - tempdb and 15-17

- tuning tools 15-18
- with consumers clause and 15-9
- worktables and 15-10
- Parameters, procedure
 - datatypes 11-8
 - optimization and 11-3
 - tuning with 10-18
- Parse and compile time 8-7
- Partition-based scans 14-5 to 14-6, 14-7
 - to 14-9, 14-12
 - asynchronous prefetch and 18-13
- partition clause, alter table command 17-22
- Partitioned tables 17-14
 - bcp (bulk copy utility) and 17-26, 23-9
 - changing the number of
 - partitions 17-23
 - command summary 17-22
 - configuration parameters for 17-17
 - configuration parameters for
 - indexing 17-24
 - create index and 15-8, 15-18, 17-24
 - creating new 17-32
 - data distribution in 17-27
 - devices and 17-28, 17-37, 17-41
 - distributing data across 17-24, 17-34
 - distribution of data 10-20
 - extent stealing and 17-29
 - information on 17-27
 - load balancing and 17-28, 17-29
 - loading with bcp 17-26
 - maintaining 17-30, 17-44
 - moving with on *segmentname* 17-33
 - parallel optimization and 14-4, 14-13
 - read-mostly 17-21
 - read-only 17-20
 - segment distribution of 17-18
 - size of 17-27, 17-30
 - skew in data distribution 10-20, 14-6
 - sorted data option and 17-32
 - space planning for 17-20
 - statistics 17-30
 - unpartitioning 17-23
 - updates and 17-21
 - updating statistics 17-30, 17-31
 - worktables 14-11
- Partitions
 - cache hit ratio and 13-23
 - guidelines for configuring 13-23
 - parallel optimization and 14-4
 - RAID devices and 13-21
 - ratio of sizes 17-27
 - size of 17-27, 17-30
- Partition statistics
 - updating 17-30
- Performance
 - backups and 23-6
 - bcp (bulk copy utility) and 23-8
 - cache hit ratio 16-10
 - clustered indexes and 3-21
 - diagnosing slow queries 14-37
 - indexes and 7-1
 - lock contention and 24-25
 - locking and 5-32
 - monitoring 24-7
 - number of indexes and 7-4
 - number of tables considered by
 - optimizer 10-8
 - run-time adjustments and 14-32
 - speed and 24-4
 - tempdb* and 19-1 to 19-14
- Performing disk I/O 21-16
- Performing network I/O 21-17
- Phantoms in transactions 5-16
- Physical device name 17-3
- Pointers
 - index 4-3
 - last page, for heap tables 3-13
 - page chain 3-5
 - text and image page 3-6
- Point query 3-2
- Pools, data cache
 - configuring for operations on heap
 - tables 3-15
 - large I/Os and 16-13
 - overhead 16-34
 - sp_sysmon report on size 24-79
- Pools, worker process 13-3
 - size 13-16

- Ports, multiple 20-13
 - Positioning `showplan` messages 9-28
 - Precedence
 - lock promotion thresholds 5-50
 - rule (execution hierarchy) 22-12, 22-13
 - Precedence rule, execution
 - hierarchy 22-13
 - Precision, datatype
 - size and 6-13
 - Predefined execution class 22-2
 - Prefetch
 - asynchronous **18-1 to 18-16**
 - data pages 10-12
 - disabling 10-13
 - enabling 10-13
 - performance ideal 16-35
 - queries 10-11
 - sequential 3-15
 - `prefetch` keyword
 - I/O size and 10-11
 - primary key constraint
 - index created by 7-6
 - Primary keys
 - normalization and 2-5
 - splitting tables and 2-14
 - Priority 22-4
 - application 22-1 to 22-29
 - assigning 22-3
 - changes, `sp_sysmon` report on 24-33, 24-35
 - precedence rule 22-13
 - run queues 22-10
 - task 22-2
 - Procedure ('proc') buffers 16-5
 - Procedure cache 21-14
 - cache hit ratio 16-6
 - errors 16-7
 - management with `sp_sysmon` 24-83
 - optimized plans in 10-27
 - query plans in 16-4
 - size report 16-5
 - sizing 16-6
 - procedure cache percent configuration
 - parameter 16-4
 - Processes (Server tasks) 21-5
 - CPUs and 24-11
 - identifier (PID) 21-4
 - lightweight 21-3
 - number of 21-4
 - overhead 21-3
 - run queue 21-5
 - Processing power 13-19
 - Process model 21-10 to 21-13
 - "proc headers" 16-6
 - Producer process 15-5, 15-19
 - Profile, transaction 24-37
 - Promotion, lock 5-47
 - `ptn_data_pgs` system function 17-30
- Q**
- `q_score_index()` routine 10-19
 - tables not listed by 10-28
 - Quantified predicate subqueries
 - aggregates in 8-30
 - optimization of 8-28
 - `showplan` messages for 9-53
 - Queries
 - execution settings 9-1
 - parallel 14-1 to 14-39
 - point 3-2
 - range 7-4
 - `showplan` setting 7-6
 - specifying I/O size 10-10
 - specifying index for 10-8
 - unindexed columns in 3-2
 - Query analysis
 - `dbcc traceon(302)` 10-17 to 10-29
 - `set noexec` 7-7
 - `set statistics io` 7-9
 - `set statistics time` 7-7
 - `showplan` and 9-1 to 9-59
 - `sp_cachestrategy` 10-15
 - tools for 7-1 to 7-53
 - Query optimizer. *See* Optimizer
 - Query plans

- cursors 12-5
- optimizer and 8-2
- procedure cache storage 16-4
- run-time adjustment of 14-31 to 14-32
- stored procedures 10-27
- sub-optimal 10-9
- triggers 10-27
- unused and procedure cache 16-4
- updatable cursors and 12-15
- Query processing
 - large I/O for 16-14
 - parallel **13-2 to 13-28**
 - steps in 8-3
- Queues
 - run 21-16
 - scheduling and 21-6
 - sleep 21-6, 21-14
 - tasks 21-11
- R**
- RAID devices
 - consumers and 15-9
 - create index and 15-9
 - partitioned tables and 13-21, 17-19
- Range
 - partition sorting 15-5
- Range queries 7-4
 - large I/O for 10-10
 - messages from 10-26
 - optimizer and 10-21
- Read-only cursors 12-6
 - indexes and 12-6
 - locking and 12-10
- Reads
 - clustered indexes and 4-6
 - disk 24-93
 - disk mirroring and 17-9
 - image* values 3-7
 - named data caches and 16-36
 - statistics for 7-15
 - text* values 3-7
- Recompilation
 - avoiding run-time adjustments 14-36
 - cache binding and 16-35
 - testing optimization and 10-18
- Recovery
 - asynchronous prefetch and 18-3
 - configuring asynchronous prefetch for 18-14
 - housekeeper task and 21-17
 - index creation and 23-4
 - log placement and speed 17-8
 - parallel sorting and 15-3, 15-18
 - sp_sysmon* report on 24-85
- recovery interval in minutes configuration
 - parameter 16-8, 16-39
 - I/O and 23-6
- Re-creating
 - indexes 17-24, 23-4
- Referential integrity
 - references and unique index requirements 7-33
 - update operations and 8-35
 - updates using 8-39
- Reformatting 8-17
 - joins and 8-17
 - parallel optimization of 15-1
 - showplan* messages for 9-35
- Relaxed LRU replacement policy
 - indexes 16-34
 - lookup tables 16-34
 - transaction logs 16-34
- Remote backups 23-6
- Replacement policy. *See* Cache replacement policy
- Replacement strategy. *See* LRU replacement strategy; MRU replacement strategy
- Replication
 - network activity from 20-9
 - tuning levels and 1-4
 - update operations and 8-35
- Reports
 - cache strategy 10-15
 - optimizer 10-19
 - procedure cache size 16-5
 - sp_estspace* 6-10

Request, client 21-16
Reserved pages, `sp_spaceused` report
 on 6-6
Resource limits 14-36
 `showplan` messages for 9-38
 `sp_sysmon` report on violations 24-36
Response time
 CPU utilization and 24-13
 definition of 1-1
 other users affecting 20-9
 parallel optimization for 14-2
 `sp_sysmon` report on 24-10
 table scans and 3-2
Retries, page splits and 24-55
Risks of denormalization 2-8
Root level of indexes 4-3
Rounding
 object size calculation and 6-12
Row ID (RID) 4-13, 24-51
 offset table 4-14
 update operations and 8-35
 updates, index maintenance
 and 24-52
 updates from clustered split 24-51
Row offset number 4-13
Rows, index
 `max_rows_per_page` and 5-35
 size of leaf 6-20
 size of non-leaf 6-21
Rows, table
 density 3-4
 `max_rows_per_page` and 5-35
 splitting 2-16
Run queue 21-4, 21-5, 21-14, 21-16, 24-27
Run-time adjustment 14-16, 14-30 to
 14-36
 avoiding 14-36
 defined 13-16
 effects of 14-32
 recognizing 14-32 to 14-36

S

Sample interval, `sp_sysmon` 24-11

Sampling for parallel sort 15-4, 15-20
SARGs. *See* Search arguments
Saturation
 CPU 13-19
 I/O 13-19
Scanning, in `showplan` messages 9-29
Scans, number of (statistics io) 7-12
Scans, table
 auxiliary scan descriptors 9-24
 avoiding 4-1
 costs of 7-18
 large I/O for 16-13
 performance issues 3-1
 `showplan` message for 9-28
Scan session 5-47
Scheduling, Server
 engines 21-11
 tasks 21-6, 21-11
Scope rule 22-13, 22-14
Search arguments 8-9
 equivalents in 8-11
 indexable 8-10
 operators in 8-10
 optimizer and 10-25
 optimizing 10-18
 parallel query optimization 14-7
 transitive closure for 8-12
 where clauses and 8-10
Search conditions
 clustered indexes and 7-32
 locking 5-5
Searches skipped, `sp_sysmon` report
 on 24-65
Second normal form 2-5
 See also Normalization
Segments 17-4
 clustered indexes on 17-12
 database object placement on 17-5,
 17-12
 free pages in 17-28
 moving tables between 17-33
 nonclustered indexes on 17-12
 parallel sorting and 15-7
 partition distribution over 17-18

- performance of parallel sort 15-17
- target 15-7, 15-19
- tempdb* 19-8
- select * command
 - logging of 19-11
- select command
 - locks created by 5-11
 - optimizing 7-3
 - parallel clause 13-15
 - specifying index 10-8
- select into/bulkcopy/pllsort database option
 - parallel sorting and 15-2
- select into command
 - heap tables and 3-13
 - large I/O for 16-14
 - parallel optimization of 14-29 to 14-30
 - in parallel queries 14-29
- Selectivity, optimizer 10-28
- Select operations
 - clustered indexes and 4-5
 - heaps 3-12
 - nonclustered indexes 4-17
- Semaphores 24-47
 - disk device contention 24-94
 - log contention 24-26
 - transaction log contention 24-26
 - user log cache requests 24-46
- Sequential prefetch 3-15, 16-13
- Serial query processing
 - deadlock detection 5-40
 - demand locks and 5-8
- Server config limit, in *sp_sysmon*
 - report 24-91
- Servers
 - monitoring performance 24-3
 - scheduler 21-7
 - uniprocessor and SMP 21-23
- Server structures 21-14
- set command
 - noexec and statistics io interaction 8-7
 - parallel degree 13-14
 - query plans 9-1 to 9-59
 - scan_parallel_degree 13-14
 - sort_resources 15-18
 - statistics io 7-11, 8-7
 - statistics time 8-7
 - subquery cache statistics 8-33
 - transaction isolation level 5-20
- Set theory operations
 - compared to row-oriented programming 12-2
- shared keyword
 - cursors and 5-27, 12-6
 - locking and 5-27
- Shared locks
 - cursors and 5-27
 - holdlock keyword 5-22
 - intent deadlocks 24-64
 - page 5-4
 - page deadlocks 24-64
 - read-only cursors 12-6
 - sp_lock report on 5-39
 - table 5-6
 - table deadlocks 24-64
- Short-circuiting subqueries 8-31
- showplan option, set 9-1 to 9-59
 - access methods 9-23
 - caching strategies 9-23
 - clustered indexes and 9-27
 - compared to trace flag 302 10-17
 - I/O cost strategies 9-23
 - messages 9-2
 - noexec and 9-1
 - query clauses 9-12
 - sorting messages 9-22
 - subquery messages 9-45
 - update modes and 9-9
- Single CPU 21-4
- Single-process overhead 21-3
- Size
 - cursor query plans 12-5
 - data pages 3-3
 - datatypes with precisions 6-13
 - formulas for tables or indexes 6-11 to 6-29
 - I/O 3-15, 16-13
 - I/O, reported by showplan 9-37
 - indexes 6-1

- nonclustered and clustered
 - indexes 4-14
- object (`sp_spaceused`) 6-4
- partitions 17-27
- predicting tables and indexes 6-14 to 6-29
- procedure cache 16-5, 16-6
- `sp_spaceused` estimation 6-9
- stored procedure 16-6
- table estimate by `dbcc` trace flags 10-20
- tables 6-1
- `tempdb` database 19-4, 19-5
- transaction logs 24-48
- triggers 16-6
- views 16-6
- Skew in partitioned tables
 - defined 14-6
 - effect on query plans 14-5
 - information on 10-20, 17-27
- Sleeping CPU 24-15
- Sleeping locks 5-38
- “sleeping” lock status 5-44
- Sleep queue 21-6, 21-14
- Slow queries 8-1
- SMP (symmetric multiprocessing)
 - systems
 - application design in 21-23
 - architecture 21-10
 - CPU use in 21-20
 - disk management in 21-24
 - fillfactor for 21-24
 - log semaphore contention 24-26
 - `max_rows_per_page` for 21-24
 - named data caches for 16-23
 - processing example 21-14
 - subsystems 21-14
 - `sysindexes` access and 24-29
 - temporary tables and 21-25
 - transaction length 21-24
- `sort_resources` option, set 15-19 to 15-22
- Sort buffers
 - computing maximum allowed 15-14
 - configuring 15-11 to 15-12
 - guidelines 15-11
 - requirements for parallel sorting 15-2
 - set `sort_resources` and 15-19
- `sorted_data` option, create index
 - partitioned tables and 17-32
 - sort suppression and 23-5
- Sorted data, reindexing 7-49, 23-4
- Sort manager. *See* Parallel sorting
- Sort operations (order by)
 - See also* Parallel sorting
 - clustered indexes and 7-26
 - covering indexes and 7-29
 - improving performance of 23-3
 - indexing to avoid 4-1
 - nonclustered indexes and 7-28
 - performance problems 19-2
 - `showplan` messages for 9-28
 - sorting plans 15-18
 - without indexes 7-25
- Sort order
 - ascending 7-27, 7-29
 - descending 7-27, 7-29
 - rebuilding indexes after
 - changing 7-48
- Sources of optimization problems 8-2
- `sp_addengine` system procedure 22-8
- `sp_addindexclass` system procedure 22-3
- `sp_bindindexclass` system procedure 22-3
- `sp_droplockpromote` system
 - procedure 5-51
- `sp_estspace` system procedure
 - advantages of 6-11
 - disadvantages of 6-11
 - planning future growth with 6-9
- `sp_helppartition` system procedure 17-27
- `sp_helpsegment` system procedure
 - checking data distribution 17-28
- `sp_lock` system procedure 5-38
- `sp_logiosize` system procedure 16-30
- `sp_monitor` system procedure
 - `sp_sysmon` interaction 24-3
- `sp_setpglockpromote` system
 - procedure 5-50
- `sp_spaceused` system procedure 6-4
 - accuracy and `dbcc` 6-9

- OAM pages and 6-3
- row total estimate reported 6-4
- sp_sysmon system procedure **24-1 to 24-98**
 - parallel sorting and 15-23
 - sorting and 15-23
 - transaction management and 24-43
- sp_who system procedure
 - blocking process 5-38
 - deadlocks and 5-42, 5-44
- Space
 - clustered compared to nonclustered indexes 4-14
 - estimating table and index size 6-14 to 6-29
 - extents 3-5
 - freeing with truncate table 16-38
 - reclaiming 3-21
 - for *text* or *image* storage 3-7
 - unused 3-5
 - worktable sort requirements 15-17
- Space allocation
 - clustered index creation 7-6
 - contiguous 3-6
 - dbcc commands for checking 6-6
 - deallocation of index pages 4-13
 - deletes and 3-14
 - extents 3-5
 - index page splits 4-9
 - monotonically increasing key values and 4-9
- Object Allocation Map (OAM)
 - pages 6-17
 - overhead calculation 6-17, 6-22
 - page splits and 4-7
 - predicting tables and indexes 6-14 to 6-29
 - procedure cache 16-6
 - sp_spaceused 6-6, 6-9
 - tempdb 19-10
 - unused space within 3-5
- Speed (Server)
 - cheap direct updates 8-36
 - deferred index deletes 8-42
 - deferred updates 8-39
 - direct updates 8-34
 - existence tests 11-2
 - expensive direct updates 8-37
 - in-place updates 8-35
 - memory compared to disk 16-1
 - select into 19-11
 - slow queries 8-1
 - sort operations 15-6, 23-3
 - updates 8-34
- Spinlocks
 - contention 16-27, 24-76
 - data caches and 16-12, 24-76
- Splitting
 - data pages on inserts 4-7
 - horizontal 2-14
 - procedures for optimization 11-4
 - tables 2-14
 - vertical 2-15
- SQL standards
 - concurrency problems 5-38
 - cursors and 12-2
- Stack (Server) 21-6
- Statistics
 - cache hits 24-70, 24-77
 - deadlocks 24-62, 24-64
 - index 10-24
 - index add levels 24-56
 - index distribution page 7-42
 - index maintenance 24-50
 - index maintenance and deletes 24-51
 - large I/O 24-71
 - locks 24-59, 24-62
 - page shrinks 24-56
 - recovery management 24-85
 - spinlock 24-76
 - subquery cache usage 8-33
 - transactions 24-40
- statistics subquerycache option, set 8-33
- status column of sp_who output 5-42
- Steps
 - deferred updates 8-39
 - direct updates 8-34
 - distribution table 7-46

- index 10-24
 - key values in distribution table 7-43
 - problem analysis 1-7
 - query plans 9-3
 - trace output for 10-24
 - update statistics and 10-26
 - values between 10-25
 - Storage management
 - collapsed tables effect on 2-12
 - delete operations and 3-14
 - I/O contention avoidance 17-5
 - page proximity 3-6
 - row density and 3-4
 - row storage 3-3
 - space deallocation and 4-11
 - Stored procedures
 - cursors within 12-8
 - hot spots and 22-22
 - network traffic reduction with 20-7
 - optimization 11-3
 - performance and 17-2
 - procedure cache and 16-4
 - query plans for 10-27
 - size estimation 16-6
 - sp_sysmon report on 24-84
 - splitting 11-4
 - temporary tables and 19-13
 - Stress tests, sp_sysmon and 24-4
 - Striping *tempdb* 19-4
 - sort performance and 15-17
 - Structures, data 21-14
 - Sub-indexes 15-5
 - Subprocesses 21-5
 - switching context 21-5
 - Subqueries
 - any, optimization of 8-28
 - attachment 8-34
 - exists, optimization of 8-28
 - expression, optimization of 8-29
 - flattening 8-28, 14-27
 - in, optimization of 8-28
 - materialization and 8-29
 - optimization 8-28, 14-27 to 14-28
 - parallel optimization of 14-27 to 14-28
 - quantified predicate, optimization
 - of 8-28
 - results caching 8-32, 14-28
 - short-circuiting 8-31
 - showplan messages for 9-45 to 9-59
 - sybsecurity* database
 - audit queue and 16-41
 - placement 17-7
 - Symbols
 - in SQL statements xlix
 - Symptoms of optimization
 - problems 8-1
 - “Sync-pt duration request” 5-39
 - “sync sleep” lock status 5-44
 - Syntax
 - conventions in manual xlix
 - sysgams* table 3-8
 - sysindexes* table
 - data access and 3-11
 - index information in 7-49
 - locking 24-29
 - sp_spaceused and 6-6
 - text objects listed in 3-7
 - sysprocedures* table
 - query plans in 16-4
 - System functions
 - OAM pages and 6-4
 - System log record, ULC flushes and (in *sp_sysmon* report) 24-45
 - System tables
 - data access and 3-11
 - performance and 17-2
 - size of 6-2
- ## T
- tablealloc option, dbcc
 - object size report 6-7
 - table count option, set 10-8
 - Table locks 5-3, 24-66
 - controlling 5-12
 - page locks vs. 5-47
 - sp_lock report on 5-39
 - types of 5-6

Tables

- collapsing 2-12
- denormalizing by splitting 2-14
- designing 2-3
- duplicating 2-13
- estimating size of 6-11 to 6-29
- heap 3-11 to 3-23
- locks held on 5-12, 5-39
- moving with on *segmentname* 17-33
- normal in *tempdb* 19-3
- normalization 2-3
- partitioning 17-14
- secondary 7-51
- size of 6-1
- size with a clustered index 6-14
- unpartitioning 17-23

Table scans

- asynchronous prefetch and 18-4
- avoiding 4-1
- cache flushing and 7-17
- evaluating costs of 7-18
- forcing 10-9
- large I/O for 16-13
- optimizer choosing 10-28
- performance issues 3-1
- row size effect on 3-4
- showplan messages for 9-26

Tabular Data Stream (TDS)

- protocol 20-3
- network packets and 24-29
- packets received 24-97
- packets sent 24-97

Target segment 15-7, 15-19**Tasks**

- client 21-2
- context switches 24-23
- CPU resources and 13-19
- demand locks and 5-7
- execution 21-16
- queued 21-6, 21-11
- scheduling 21-6, 21-11
- sleeping 24-27

***tempdb* database**

- data caches 19-10

- logging in 19-11
- named caches and 16-13
- performance and 19-1 to 19-14
- placement 17-6, 19-8
- segments 19-8
- size of 19-5
- in SMP environment 21-25
- space allocation 19-10
- striping 19-4

Temporary tables

- denormalization and 19-5
- indexing 19-13
- nesting procedures and 19-14
- normalization and 19-5
- optimizing 19-12
- performance considerations 17-2, 19-2
- permanent 19-3
- SMP systems 21-25

Testing

- caching and 7-16
- data cache performance 16-10
- “hot spots” 7-32
- index forcing 10-9
- nonclustered indexes 7-34
- performance monitoring and 24-3
- statistics io and 7-16

***text* datatype**

- chain of text pages 6-29
- large I/O for 16-14
- page number estimation 6-28 to 6-29
- page size for storage 3-7
- storage on separate device 3-6, 17-13
- sysindexes* table and 3-7

Third normal form. *See* Normalization**Thresholds**

- bulk copy and 23-8
- database dumps and 23-6
- transaction log dumps and 17-7

Throughput 1-1

- adding engines and 24-14
- CPU utilization and 24-13
- group commit sleeps and 24-27
- log I/O size and 24-27

- measuring for devices 17-19
 - monitoring 24-10
 - pool turnover and 24-78
 - TDS messages and 24-98
- Time interval
 - deadlock checking 5-31
 - recovery 16-40
 - since `sp_monitor` last run 21-19
 - `sp_sysmon` 24-5
- Time slice 22-3
- time slice configuration parameter 21-7
 - CPU yields and 21-9
- Tools
 - index information 7-7
 - packet monitoring with
 - `sp_monitor` 20-6
 - query analysis 7-6
- Total cache hits in `sp_sysmon`
 - report 24-70
- Total cache misses in `sp_sysmon` report
 - on 24-70
- Total cache searches in `sp_sysmon`
 - report 24-70
- Total disk I/O checks in `sp_sysmon`
 - report 24-16
- Total lock requests in `sp_sysmon`
 - report 24-62
- total memory configuration
 - parameter 16-2
- Total network I/O checks in `sp_sysmon`
 - report 24-15
- Total work compared to response time
 - optimization 14-2
- Trace flag 302 10-17 to 10-29
- transaction isolation level option, set 5-20
- Transaction logs
 - average writes 24-48
 - cache replacement policy for 16-34
 - contention 24-26
 - I/O batch size 24-25
 - last page writes 24-27
 - log I/O size and 16-29
 - named cache binding 16-13
 - page allocations 24-48
 - placing on separate segment 17-7
 - on same device 17-8
 - semaphore contention 24-26
 - storage as heap 3-22
 - task switching and 24-27
 - update operation and 8-34
 - writes 24-48
- Transactions
 - close on `endtran` option 5-27
 - committed 24-38
 - deadlock resolution 5-30
 - default isolation level 5-20
 - locking 5-3
 - logging and 19-11
 - log records 24-44, 24-46
 - management 24-43
 - monitoring 24-10
 - multidatabase 24-40, 24-45
 - performance and 24-10
 - profile (`sp_sysmon` report) 24-37
 - SMP systems 21-24
 - statistics 24-40
- Transitive closure for SARGs 8-12
- Triggers
 - managing denormalized data
 - with 2-17
 - procedure cache and 16-4
 - query plans for 10-27
 - `showplan` messages for 9-37
 - size estimation 16-6
 - update operations and 8-35
- TRUE, return value of 8-28
- `truncate table` command
 - distribution pages and 10-27
 - not allowed on partitioned
 - tables 17-17
- `tsequal` system function
 - compared to `holdlock` 5-38
- Tuning
 - advanced techniques for 10-1 to 10-29
 - asynchronous prefetch 18-10
 - definition of 1-2
 - levels 1-3 to 1-7
 - monitoring performance 24-3

- parallel query 13-22
 - parallel query processing 13-18 to 13-25
 - parallel sorts 15-10 to 15-18
 - range queries 10-9
 - recovery interval 16-40
 - trace flag 302 for 10-18 to 10-29
 - Turnover, pools (sp_sysmon report on) 24-78
 - Turnover, total (sp_sysmon report on) 24-80
 - Two-phase commit
 - network activity from 20-8
- U**
- ULC. *See* User log cache (ULC)
 - union operator
 - cursors and 12-15
 - optimization of joins using 11-5
 - parallel optimization of 14-28, 15-1
 - subquery cache numbering and 8-33
 - Uniprocessor system 21-4
 - Unique constraints
 - index created by 7-6
 - Unique indexes 4-1
 - optimizer choosing 10-23
 - optimizing 7-33
 - update modes and 8-43
 - Units, allocation. *See* Allocation units
 - Unknown values, optimizing 10-26
 - unpartition clause, alter table 17-23
 - Unpartitioning tables 17-23
 - Unused space
 - allocations and 3-5
 - update command
 - image* data and 6-29
 - locks created by 5-11
 - text* data and 6-29
 - transaction isolation levels and 5-16
 - Update cursors 12-6
 - Update locks 5-5
 - cursors and 12-6
 - deadlocks and 5-30
 - sp_lock report on 5-39
 - Update modes
 - cheap direct 8-36
 - deferred 8-39
 - deferred index 8-39
 - expensive direct 8-37, 8-38
 - indexing and 8-43
 - in-place 8-35
 - optimizing for 8-42
 - Update operations 8-34
 - checking types 24-42
 - heap tables and 3-14
 - “hot spots” 5-34
 - index maintenance and 24-50
 - index updates and 7-35
 - scan counts and 7-14
 - statistics 24-42
 - Update page deadlocks, sp_sysmon report on 24-64
 - update partition statistics command 17-30, 17-31
 - update statistics command
 - distribution pages and 7-42
 - large I/O for 16-14
 - steps and 10-26
 - User connections
 - application design and 24-23
 - network packets and 20-4
 - sp_sysmon report on 24-23
 - User-defined execution class 22-3
 - User log cache (ULC)
 - log records 24-44, 24-46
 - log semaphore contention and 24-26
 - log size and 16-29
 - maximum size 24-46
 - semaphore requests 24-46
 - user log cache size configuration
 - parameter 24-46
 - increasing 24-45
 - Users
 - assigning execution priority 22-22
 - logins information 21-12
 - Utilization
 - cache 24-76

- engines 24-12
- kernel 24-11

V

Values

- unknown, optimizing 10-26, 11-3

Variable-length columns

- index overhead and 7-51
- row density and 3-4

Variables

- optimizer and 10-27, 11-3

Vertical table splitting 2-15

Views

- collapsing tables and 2-13
- size estimation 16-6

W

Wash area 16-8

- configuring 16-33
- parallel sorting and 15-16

Wash marker 3-16

where clause

- creating indexes for 7-32
- evaluating 10-21
- optimizing 10-18
- search arguments and 8-10
- table scans and 3-12

with consumers clause, create index 15-9

Worker processes 13-3, 21-2

- clustered indexes and 15-8
- configuring 13-14
- consumer process 15-5
- coordinating process 15-6
- deadlock detection and 5-42
- joins and 14-18
- locking and 5-9
- nonclustered indexes and 15-8
- overhead of 14-3
- parallel sorting and 15-9
- parallel sort requirements 15-6
- pool 13-3
- pool size and 13-16

- producer process 15-5
- resource limits with 14-39
- run-time adjustment of 14-16, 14-30
 - to 14-36
- worktable sorts and 15-10

Worktables

- distinct and 9-19
- locking and 19-11
- or clauses and 8-24
- order by and 9-21
- parallel queries and 14-11, 14-28
- parallel sorting and 15-10, 15-12
- parallel sorts on 14-28
- partitioning of 14-11
- reads and writes on 7-16
- reformatting and 8-17
- showplan messages for 9-13
- space requirements 15-17
- statistics for I/O 7-15
- tempdb* and 19-3

Write operations

- contention 24-25
- disk 24-93
- disk mirroring and 17-9
- free 21-17
- housekeeper process and 21-18
- image* values 3-7
- serial mode of disk mirroring 17-10
- statistics for 7-15
- text* values 3-7
- transaction log 24-48

Y

Yields, CPU

- cpu grace time configuration
 - parameter 21-9
- sp_sysmon report on 24-14
- time slice configuration parameter 21-9
- yield points 21-8

