Sybase® Adaptive Server™ Enterprise Performance and Tuning Guide: Query Tuning

Adaptive Server Enterprise Version 12

Document ID: 32614-01-1200-02

Last Revised: October 1999

Principal author: Karen Paulsell

Contributing authors: Server Publications Group

Document ID: 32614-01-1200-02

This publication pertains to Adaptive Server Enterprise Version 12 of the Sybase database management software and to any subsequent version until otherwise indicated in new editions or technical notes. Information in this document is subject to change without notice. The software described herein is furnished under a license agreement, and it may be used or copied only in accordance with the terms of that agreement.

Document Orders

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

Customers in other countries with a U.S. license agreement may contact Customer Fulfillment via the above fax number. All other international customers should contact their Sybase subsidiary or local distributor.

Upgrades are provided only at regularly scheduled software release dates.

Sybase Trademarks

Restricted Rights

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

Table of Contents

About This Book

Basic Concepts

1. Introduction to Performance Analysis

2. Database Design and Denormalizing for Performance

3. Data Storage

4. How Indexes Work

Tuning Query Performance

5. Understanding the Query Optimizer

6. Access Methods and Query Costing for Single Tables

7. Access Methods and Query Costing for Joins and Subqueries

8. Cursors and Performance

9. Indexing for Performance

10. Managing Statistics to Improve Performance

Parallel Query Concepts and Tuning

11. Introduction to Parallel Query Processing

12. Parallel Query Optimization

13. Parallel Sorting

Query Tuning Tools

14. Introduction to Query Tuning Tools

15. Determining or Estimating the Sizes of Tables and Indexes

16. Using the *set statistics* Commands

17. Using *set showplan*

18. Tuning with *dbcc traceon*

19. Statistics Tables and Displaying Statistics with *optdiag*

20. Advanced Optimizing Tools

Query Tuning With Abstract Plans

21. Introduction to Abstract Plans

22. Creating and Using Abstract Plans

23. Managing Abstract Plans with System Procedures

24. Abstract Plan Language Reference

25. A User's Guide to Abstract Query Plans

List of Figures

List of Tables

About This Book

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

Audience

This manual is intended for:

- Sybase System Administrators
- Database designers
- Application developers

How to Use This Book

The *Performance and Tuning Guide* is divided into these volumes:

- *Performance and Tuning: Query Tuning*
- *Performance and Tuning: System Tuning*

Performance and Tuning: Query Tuning

This book contains the following parts and chapters:

- Part 1, Basic Concepts
  - 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.

- Part 2, Tuning Query Performance
  - Chapter 5, “Understanding the Query Optimizer,” explains the process of query optimization, and how statistics are applied to search arguments and joins for queries.
  - Chapter 6, “Access Methods and Query Costing for Single Tables,” describes how Adaptive Server accesses tables in queries that only involve a single table, and how the costs are estimated for various access methods.
  - Chapter 7, “Access Methods and Query Costing for Joins and Subqueries,”describes how Adaptive Server accesses tables during joins and subqueries and how the costs are determined.
  - Chapter 8, “Cursors and Performance,” describes performance issues with cursors.
  - Chapter 9, “Indexing for Performance,” provides guidelines and examples for choosing indexes.
  - Chapter 10, “Managing Statistics to Improve Performance,” describes how to use the **update statistics** command to create and update statistics.
- Part 3, Parallel Query Concepts and Tuning
  - Chapter 11, “Introduction to Parallel Query Processing,” introduces the concepts and resources required for parallel query processing.
  - Chapter 12, “Parallel Query Optimization,” provides an in-depth look at the optimization of parallel queries.
  - Chapter 13, “Parallel Sorting,” describes the use of parallel sorting for queries and for creating indexes.
- Part 4, Query Tuning Tools
  - Chapter 14, “Introduction to Query Tuning Tools,” presents an overview of query tuning tools and describes how these tools can interact.
  - Chapter 15, “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 16, “Using the set statistics Commands,” explains the commands that provide information about query execution.
  - Chapter 17, “Using set showplan,” provides examples of **showplan** messages.

- Chapter 18, “Tuning with dbcc traceon,” explains how to use the **dbcc traceon** commands to analyze query optimization problems.
- Chapter 19, “Statistics Tables and Displaying Statistics with optdiag,” describes the tables that store statistics and the output of the **optdiag** command that displays the statistics used by the query optimizer.
- Chapter 20, “Advanced Optimizing Tools,” describes advanced tools for tuning query performance.

• Part 5, Query Tuning With Abstract Plans
  - Chapter 21, “Introduction to Abstract Plans,” provides an overview of abstract plans and how they can be used to solve query optimization problems.
  - Chapter 22, “Creating and Using Abstract Plans,” describes the commands that can be used to save and use abstract plans.
  - Chapter 23, “Managing Abstract Plans with System Procedures,” describes the system procedures that manage abstract plans and abstract plan groups.
  - Chapter 24, “Abstract Plan Language Reference,” describes the abstract plan language.
  - Chapter 25, “A User’s Guide to Abstract Query Plans,” provides an introduction to writing abstract plans for specific types of queries and to using abstract plans to detect changes in query optimization due to configuration or system changes.

Performance and Tuning: System Tuning

• Part 6, Locking
  - Chapter 26, “Locking in Adaptive Server,” describes the types of locks that Adaptive Server uses and what types of locks are acquired during query processing.
  - Chapter 27, “Locking Commands,” describes the commands that set locking schemes for tables and control isolation levels and other locking behavior during query processing.
  - Chapter 28, “Reporting on Locks,” describes the system procedures that report on locks and lock contention.
  - Chapter 29, “Locking Configuration and Tuning,” describes the impact of locking on performance and describes the tools to analyze locking problems and configure locking.

- Part 7, Application Maintenance
  - Chapter 30, "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 31, "Setting Space Management Properties," describes how space management properties can be set for tables to improve performance and reduce the frequency of maintenance operations on tables and indexes.
- System-Level Tuning
  - Chapter 32, "Memory Use and Performance," describes how Adaptive Server uses memory for the procedure and data caches.
  - Chapter 33, "Controlling Physical Data Placement," describes the uses of segments and partitions for controlling the physical placement of data on storage devices.
  - Chapter 34, "Tuning Asynchronous Prefetch," describes how asynchronous prefetch improves performance for queries that perform large amounts of disk I/O.
  - Chapter 35, "tempdb Performance Issues," stresses the importance of the temporary database, *tempdb*, and provides suggestions for improving its performance.
  - Chapter 37, "How Adaptive Server Uses Engines and CPUs," describes how client processes are scheduled on engines in Adaptive Server.
  - Chapter 38, "Distributing Engine Resources Between Tasks," describes how to assign execution precedence to specific applications.
  - Chapter 39, "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 the Sybase Technical Library.

- The Adaptive Server installation documentation for your platform – describes installation, upgrade, and configuration procedures for all Adaptive Server and related Sybase products.
- *What's New in Adaptive Server Enterprise?* – Describes the new features in Adaptive Server version 12, the system changes added to support those features, and the changes that may affect your existing applications.
- *Transact-SQL User's Guide* – Documents Transact-SQL, Sybase's enhanced version of the relational database language. This manual serves as a textbook for beginning users of the database management system. This manual also contains descriptions of the *pubs2* and *pubs3* sample databases.
- *System Administration Guide* – Provides in-depth information about administering servers and databases. This manual includes instructions and guidelines for managing physical resources, security, user and system databases, and specifying character conversion, international language, and sort order settings.
- *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.
- *Error Messages* and *Troubleshooting Guide* – Explains how to resolve frequently occurring error messages and describes solutions to system problems frequently encountered by users.
- *Component Integration Services User's Guide* – Explains how to use the Adaptive Server Component Integration Services feature to connect remote Sybase and non-Sybase databases.

- *Java in Adaptive Server Enterprise* – Describes how to install and use Java classes as datatypes and user-defined functions in the Adaptive Server database.
- *Using Sybase Failover in a High Availability System* – Provides instructions for using Sybase's Failover to configure an Adaptive Server as a companion server in a high availability system.
- *Using Adaptive Server Distributed Transaction Management Features* – Explains how to configure, use, and troubleshoot Adaptive Server DTM Features in distributed transaction processing environments.
- *XA Interface Integration Guide for CICS, Encina, and TUXEDO* Provides instructions for using Sybase's DTM XA Interface with X/Open XA transaction managers.
- *Adaptive Server Glossary* – Defines technical terms used in the Adaptive Server documentation.

Other Sources of Information

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

- Technical Library CD contains product manuals and technical documents and is included with your software. The DynaText browser (included on the Technical Library CD) allows you to access technical information about your product in an easy-to-use format.

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

- Technical Library Web site includes the Product Manuals site, which is an HTML version of the Technical Library CD that you can access using a standard Web browser. In addition, you'll find links to the Technical Documents Web site (formerly known as Tech Info Library), the Solved Cases page, and Sybase/Powersoft newsgroups.

  To access the Technical Library Web site, go to support.sybase.com, click the Electronic Support Services tab, and select a link under the Technical Library heading.

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 *italics*.	*master* database
Variables, or words that stand for values that you fill in, are in *italics*.	`select` *column_name* `from` *table_name* `where` *search_conditions*
Parentheses are to be typed as part of the command.	`compute` *row_aggregate* (*column_name*)
Curly braces indicate that you must choose at least one of the enclosed options. Do not type the braces.	`{cash, check, credit}`
Brackets mean choosing one or more of the enclosed options is optional. Do not type the brackets.	`[anchovies]`
The vertical bar means you may select only one of the options shown.	`{die_on_your_feet | live_on_your_knees |` `live_on_your_feet}`
The comma means you may choose as many of the options shown as you like, separating your choices with commas to be typed as part of the command.	`[extra_cheese, avocados, sour_cream]`

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

Element	Example
An ellipsis (...) means that you can **repeat** the last unit as many times as you like.	`buy thing = price [cash	check	credit]` ` [, thing = price [cash	check	credit]]...` 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.

- 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 19, "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
expression	Can include constants, literals, functions, column identifiers, variables, or parameters
logical expression	An expression that returns TRUE, FALSE, or UNKNOWN
constant expression	An expression that always returns the same value, such as “5+3” or “ABCDE”
float_expr	Any floating-point expression or expression that implicitly converts to a floating value
integer_expr	Any integer expression, or an expression that implicitly converts to an integer value
numeric_expr	Any numeric expression that returns a single value
char_expr	Any expression that returns a single character-type value
binary_expression	An expression that returns a single *binary* or *varbinary* 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"?

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.

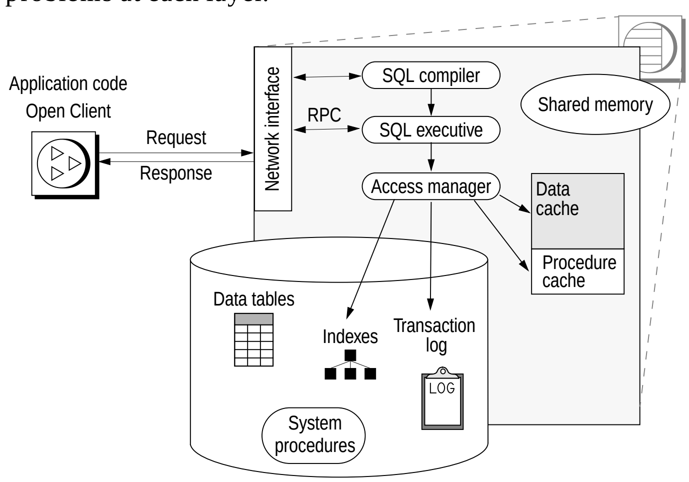


Figure 1-1: Adaptive Server system model

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, 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:

- Application 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, and data cache.
- Server layer – At the server layer, there are many shared resources, including the data and procedure caches, locks, and 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(DSS) 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 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 to address these issues include:

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

Database Layer

Issues at the database layer include:

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

Options to address these issues include:

- Using transaction log thresholds to automate log dumps and avoid running out of space
- Using thresholds for space monitoring in data segments
- Using partitions to speed loading of data

- Placing objects on devices to avoid disk contention or to take advantage of I/O 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 to address these issues include:

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

Devices Layer

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 to address these issues include:

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

Network Layer

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

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

Options to address these issues include:

- Configuring packet sizes to match application needs
- Configuring subnets
- Isolating heavy network uses
- Moving to 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 to address these issues include:

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

- Configuring multiple data caches

Operating System Layer

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 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 39, "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 is correct, implement the solution in your development environment.

2 Database Design and Denormalizing for Performance

This 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. If you start with a bad database design this book may help you speed up your queries a little, but good overall performance starts with good 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. It cannot teach you nearly as much as the many excellent books or classes available on relational database design.

This chapter contains the following sections:

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).

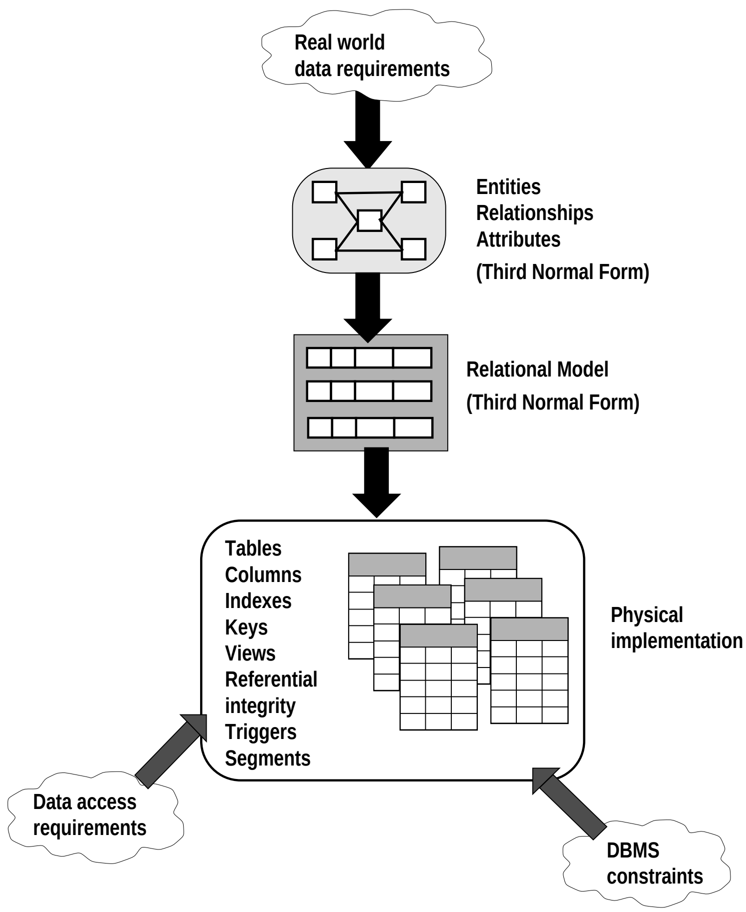


Figure 2-1: Database design

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 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.

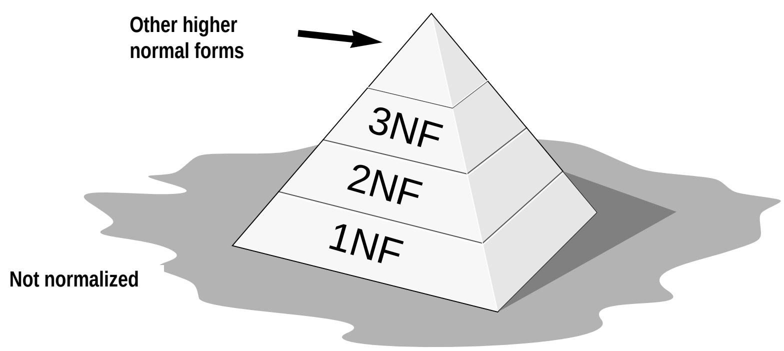


Figure 2-2: Levels of normalization

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.
- Each 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)

Employee

emp_num	emp_lname
10052	Jones
10101	Sims

Emp_dept (emp_num, dept_no)

Emp_dept

emp_num	dept_no
10052	A10
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 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

Depends on part of primary key (dept_name)

Primary key (emp_num, dept_no)

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 (emp_num, dept_no)

Dept (dept_no, dept_name)

Dept

dept_no	dept_name
A10	accounting
D60	development

Primary key (dept_no)

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)

Dept

dept_no	dept_name	mgr_emp_num	mgr_lname
A10	accounting	10073	Johnson
D60	development	10089	White
M80	marketing	10035	Dumont

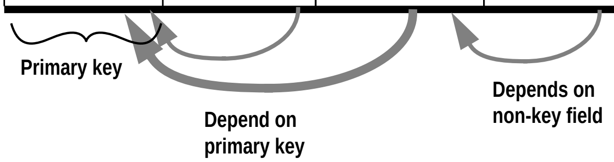


Figure 2-7: A table that violates Third Normal Form

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 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 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
```

Obtaining the summary values and the document title requires a join with the *titles* table:

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

If you run this query frequently, it makes sense to denormalize this table. But there is a price to pay: you must create an insert/update/delete trigger on the *salesdetail* table to maintain the aggregate values in the *titles* table. Executing the trigger and performing the changes to *titles* adds processing cost to each data modification of the *qty* column value in *salesdetail*.

This situation is a good example of the tension between decision support applications, which frequently need summaries of large amounts of data, and transaction processing applications, which perform discrete data modifications. Denormalization usually favors one form of processing at a cost to others.

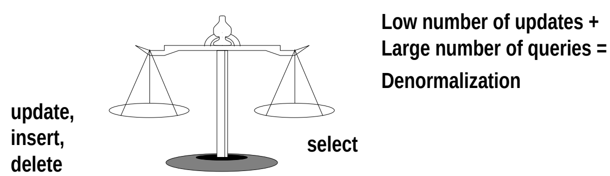


Figure 2-9: Balancing denormalization issues

Whatever form of denormalization you choose, it has the potential for data integrity problems that you must document carefully and address 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 reevaluated if the application changes.
- It can increase the size of tables.
- In some instances, it simplifies coding; in others, it makes coding more complex.

Performance Advantages 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.*

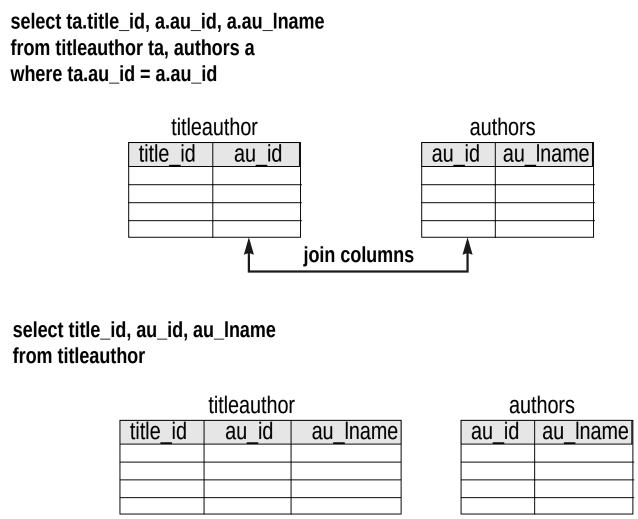


Figure 2-10: Denormalizing by adding redundant columns

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 eliminate some joins and reduce the time needed to produce aggregate values. The *total_sales* column in the *titles* table of the *pubs2* database provides one example of a derived column used to reduce aggregate value processing time.

The example in Figure 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.

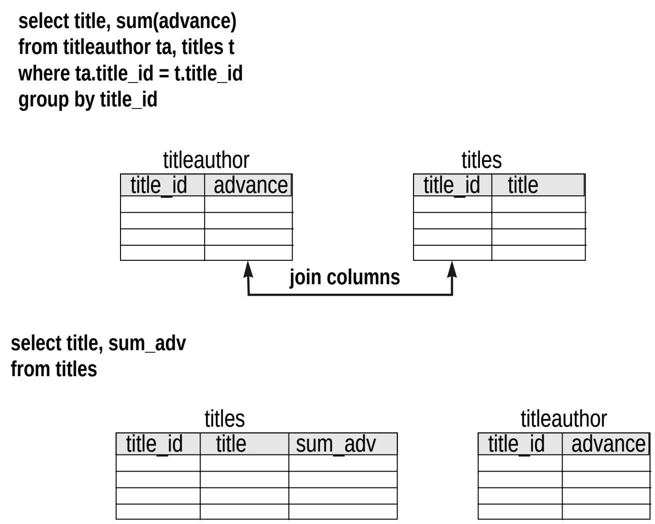


Figure 2-11: Denormalizing by adding derived columns

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.

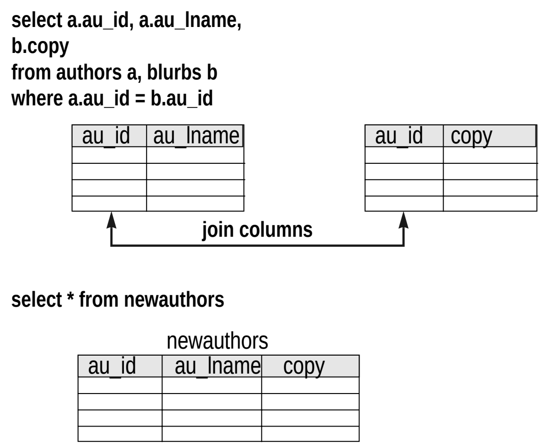


Figure 2-12: Denormalizing by collapsing tables

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.

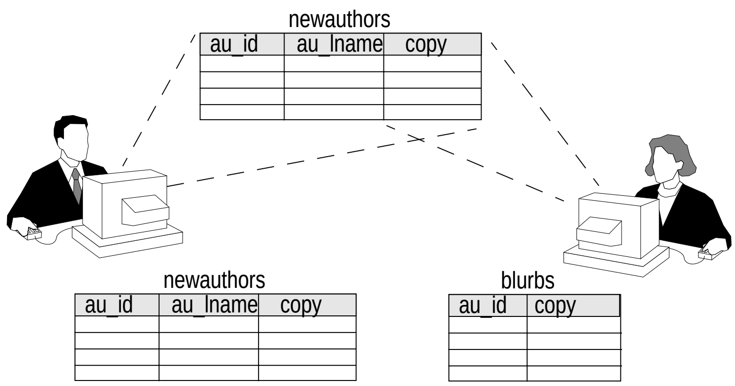


Figure 2-13: Denormalizing by duplicating tables

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

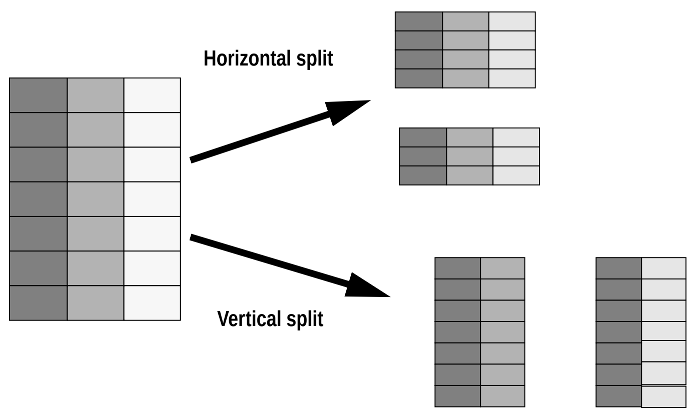


Figure 2-14: Horizontal and vertical partitioning of tables

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 requires different table names in queries, depending on values in the tables. This complexity 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:

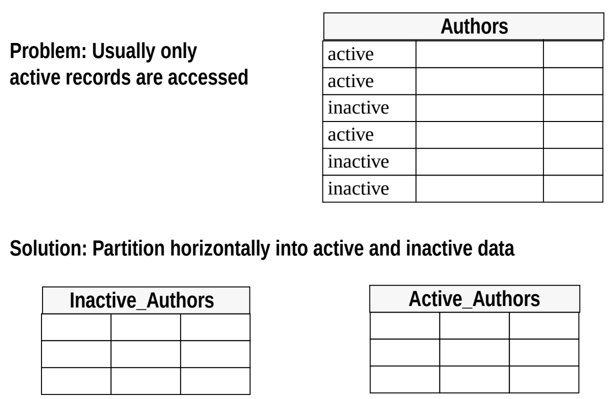


Figure 2-15: Horizontal partitioning of active and inactive data

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 accessed infrequently, placing them in a separate table can greatly speed the retrieval of the more frequently used columns. With shorter rows, more data rows fit on a data page, so, for many queries, fewer pages can be accessed.

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

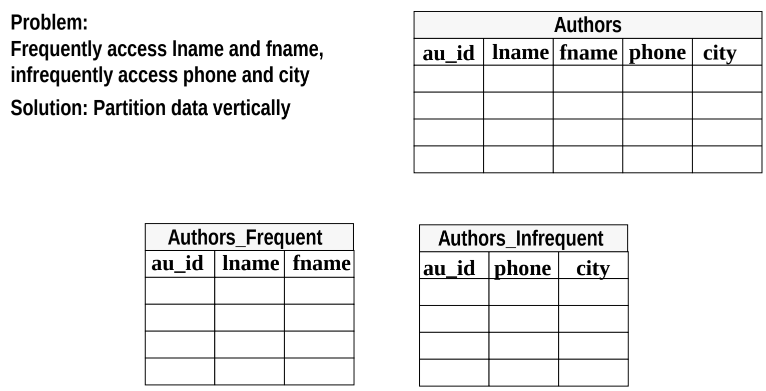


Figure 2-16: Vertically partitioning a table

Managing Denormalized Data

Whatever denormalization techniques you use, you need to ensure data integrity. Techniques you can use 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 ensure that changes are atomic
- Batch reconciliation, run at appropriate intervals, to bring the denormalized data back into agreement

From an integrity point of view, triggers provide the best solution, although they can be costly in terms of performance.

Using Triggers 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.

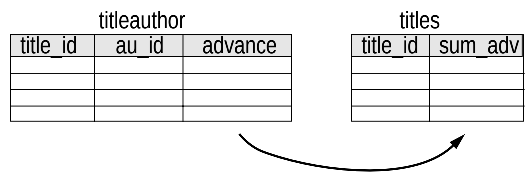


Figure 2-17: Using triggers to maintain normalized data

Using Application Logic to Manage Denormalized Data

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

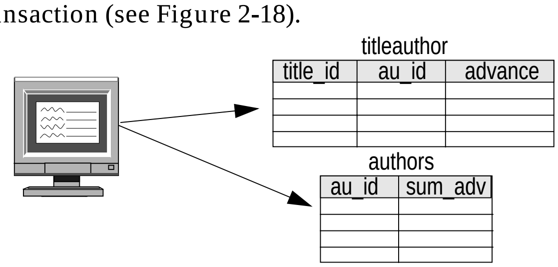


Figure 2-18: Maintaining denormalized data via application logic

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, infrequent batches (see Figure 2-19).

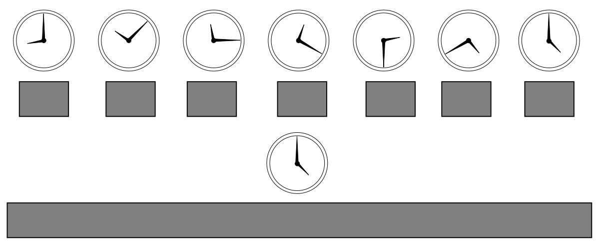


Figure 2-19: Using batch reconciliation to maintain data

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

Each time you submit a Transact-SQL query, Adaptive Server's 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 as few as 3 or 4 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.

Query Processing and Page Reads

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

When a query performs a table scan, Adaptive Server reads every page in the table because no useful indexes are available to help it retrieve the data. The individual query may have poor response time, because disk reads take time. Queries that incur costly table scans also affect the performance of other queries on your server. Table scans can increase the time other users have to wait for a response, since they consume system resources such as CPU time, disk I/O, and network capacity.

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

This chapter describes how tables are stored, and how access to data rows takes place when indexes are not being used. Chapter 4, “How Indexes Work,” describes access methods for indexes. 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. These two chapters provide a basis for understanding how the optimizer models the cost of accessing the data for your queries.

Adaptive Server Pages

The basic unit of storage for Adaptive Server is a **page**. A page is 2K, 2048 bytes. These types of pages store database objects:

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

Page Headers and Page Sizes

All pages have a header that stores information such as the object ID that the page belongs to and other information used to manage space on the page. Table 3-1 shows the number of bytes of overhead and usable space on data and index pages.

Table 3-1: Overhead and user data space on data and index pages

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

The rest of the page is available to store data and index rows. For information on how text, image, and Java columns are stored, see “Large Object (LOB) Pages” on page 3-4.

Data and Index Pages

Data pages and index pages on data-only-locked tables have a row offset table that stores pointers to the starting byte for each row on the page. Each pointer takes 2 bytes. Figure 3-1 shows a data page.

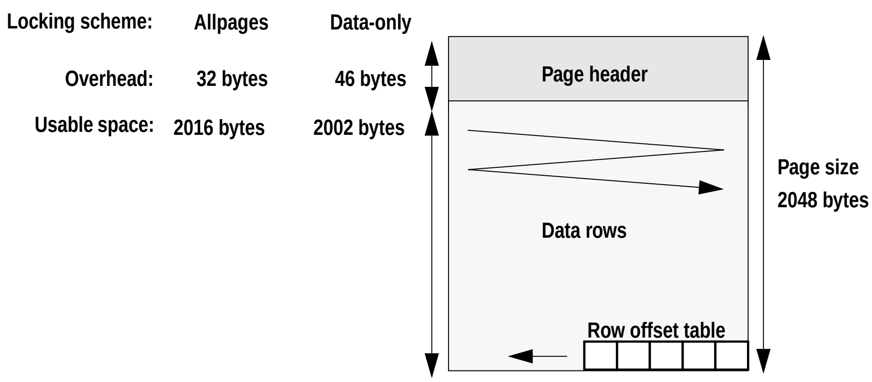


Figure 3-1: An Adaptive Server data page

Data and index rows are inserted on a page starting just after the page header, and fill contiguously down the page. For all tables and indexes on data-only-locked tables, the row offset table begins at the last byte on the page, and grows upward. The information stored for each row consists of the actual column data plus information such as the row number and the number of variable-length and null columns in the row. Index pages for allpages-locked tables do not have a row offset table.

Rows cannot cross page boundaries, except for *text*, *image*, and Java off-row columns. Each data row has at least 4 bytes of overhead; rows that contain variable-length data have additional overhead. See Chapter 15, "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.

Large Object (LOB) Pages

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

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

Figure 3-2 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.

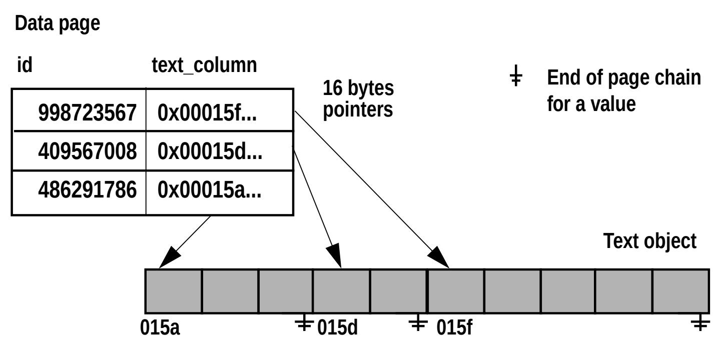


Figure 3-2: Large object storage

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

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

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

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

Extents

Adaptive Server pages are always allocated to a table, index, or LOB structure 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 pages. Extents are deallocated only when all the pages in an extent are empty. Table 3-3 shows 8 data pages in an extent.

Figure 3-3: An extent of 8 data pages

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

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

In this report, the *titles* table and its indexes have 1392K reserved on various extents, including 48K (24 data pages) unallocated in those extents.

Pages That Manage Space Allocation

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

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

The following types of pages track the disk space use by database objects:

- Global allocation map (GAM) pages contain allocation bitmaps for an entire database.
- Allocation pages track space usage and objects within groups of 256 pages, or 1/2MB.
- Object allocation map (OAM) pages contain information about the extents used for an object. Each table and index has at least

one OAM page that tracks where pages for the object are stored in the database.

- Control pages manage space allocation for partitioned tables. Each partition has one control page.

Global Allocation Map (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. Page 0 and all pages that are multiples of 256 are allocation pages.

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

Object Allocation Map (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. OAM pages store pointers to the allocation units that contain pages for the object.

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

How OAM Pages and Allocation Pages Manage Object Storage

Figure 3-4 shows how allocation units, extents, and objects are managed by OAM pages and allocation pages.

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

allocation page 0 points to page 1 and 24 for the table, and allocation page 256 points to pages 272 and 504.

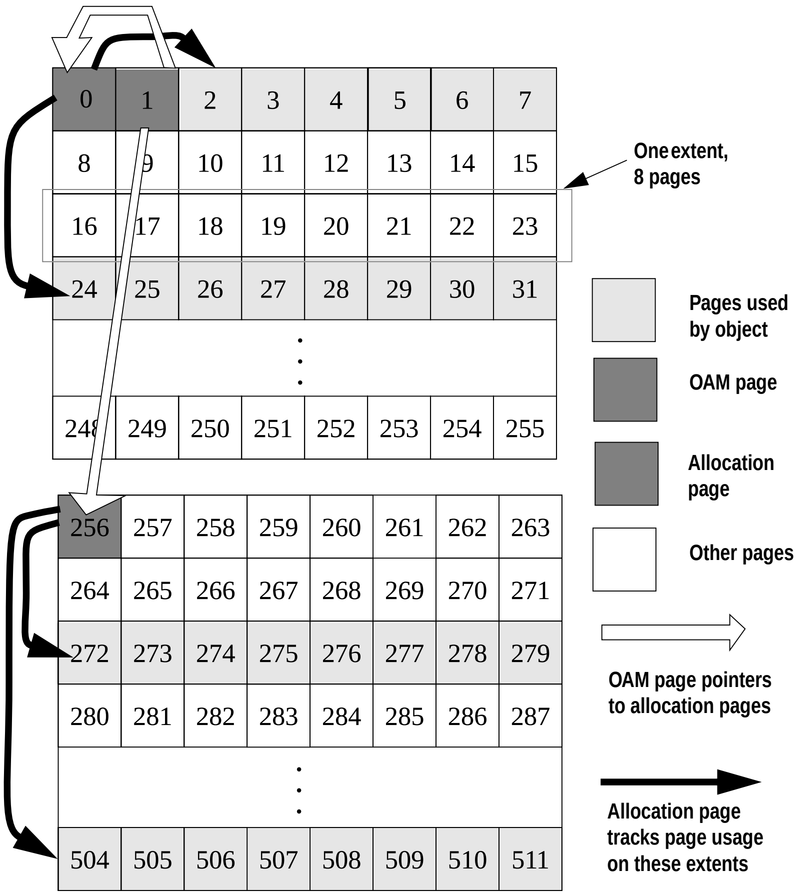


Figure 3-4: OAM page and allocation page pointers

Page Allocation Keeps an Object's Pages Together

Adaptive Server tries to keep the page allocations close together for objects. In most cases:

- If there is an unallocated page in the current extent, that page is assigned to the object.

- If there is no free page in the current extent, but there is an unallocated page on another of the object's extents, that extent is used.
- If all the object's extents are full, but there are free extents on the allocation unit, the new extent is allocated in a unit already used by the object.

The *sysindexes* Table and Data Access

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

- For each allpages-locked table, the *indid* column is 0 if the table does not have a clustered index, and 1 if the table does have a clustered index.
- For data-only-locked tables, the *indid* is always 0 for the table.
- For each nonclustered index, and for each clustered index on a data-only-locked table, index IDs are between 2 and 250.
- For each table with one or more LOB columns, the index ID is always 255 for the LOB structure.

Each row in *sysindexes* stores pointers to a table or index to speed access to objects. Table 3-2 shows how these pointers are used during data access.

Table 3-2: Use of sysindexes pointers in data access

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

Heaps of Data: Tables Without Clustered Indexes

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

The phrase “no useful index” is important in describing the optimizer’s decision to perform a table scan. Sometimes, an index exists on the columns named in a **where** clause, but the optimizer determines that it would be more costly to use the index than to perform a table scan. 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 always used when you select all rows in a table. The only exception is when the query includes only columns that are keys in a nonclustered index. For more information, see “Index Covering” on page 4-21.

The following sections describe how Adaptive Server locates rows when a table has no useful index.

Lock Schemes and Differences Between Heaps

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

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

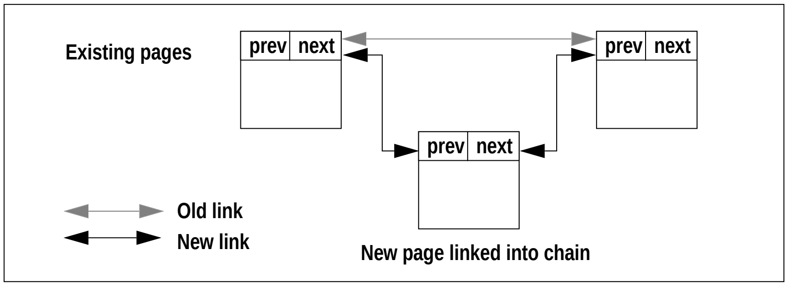


Figure 3-5: Page linkage in allpages-locked tables

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

Another difference between allpages-locked tables and data-only-locked tables is that data-only-locked tables used fixed row IDs. This means that row IDs (a combination of the page number and the row number on the page) do not change in a data-only-locked table during normal query processing. Row IDs change only when one of the operations that require data-row copying is performed, for example, during **reorg rebuild** or while creating a clustered index. For information on how fixed row IDs affect heap operations, see "Deleting from a Data-Only Locked Heap Table" on page 3-16 and "Updates on Data-Only-Locked Heap Tables" on page 3-18.

Select Operations on Heaps

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

Allpages-Locked Heap Tables

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

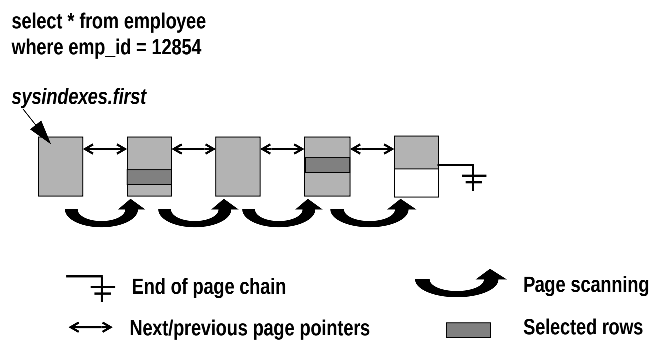


Figure 3-6: Selecting from an allpages-locked heap

Data-Only Locked Heap Tables

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

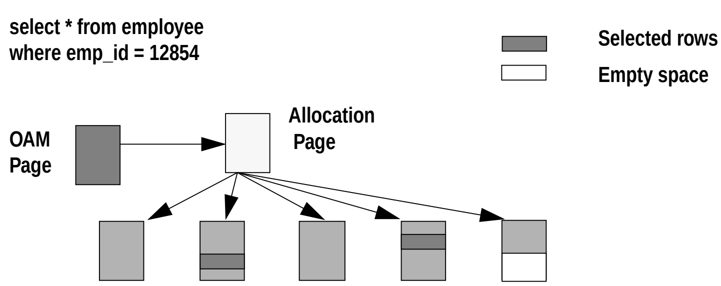


Figure 3-7: Selecting rows from a data-only-locked table

Inserting Data into an Allpages-Locked Heap Table

When you insert data into an allpages-locked heap table, the data row is always added to the last page of the table. If there is no clustered index on a table, and the table is not partitioned, the *sysindexes.root* entry for the heap table stores a pointer to the last page of the heap to locate the page where the data needs to be inserted. If the last page is full, a new page is allocated in the current extent and linked onto the chain. If the extent is full, Adaptive Server looks for empty pages on other extents being used by the table. If no pages are available, a new extent is allocated to the table.

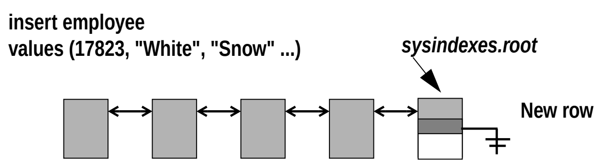


Figure 3-8: Inserting a row into an allpages-locked heap table

Contention During Heap Inserts

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

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

Some workarounds for last-page contention on heaps include:

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

Other guidelines, that apply to all transactions where there may be lock contention include:

- Keeping transactions short
- Avoiding network activity and user interaction, whenever possible, once a transaction acquires locks

Inserting Data into a Data-Only-Locked Heap Table

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

Blocking while many users are simultaneously inserting data is much less likely to occur during inserts to data-only-locked heap tables. When blocking occurs, Adaptive Server allocates a small number of empty pages and directs new inserts to those pages using these newly allocated pages as hints. For datarows-locked tables, blocking occurs only while the actual changes to the data page are being written; although row locks are held for the duration of the transaction, other rows can be inserted on the page. The row-level locks allow multiple transaction to hold locks on the page. Figure 3-9 shows a data-only-locked table with several hints directing inserts to different pages.

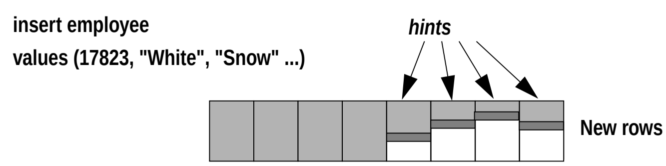


Figure 3-9: Hints direct inserts to pages with space

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

If Contention Occurs During Heap Inserts

Contention during inserts to heap tables is greatly reduced for data-only-locked tables, but can still take place. If contention slows inserts, some workarounds include:

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

Deleting Data from a Heap Table

When you delete rows from a heap table, and there is no useful index, Adaptive Server scans the data rows in the table to find the rows to delete. It has no way of knowing how many rows match the conditions in the query without examining every row.

Deleting from an Allpages-Locked Heap Table

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

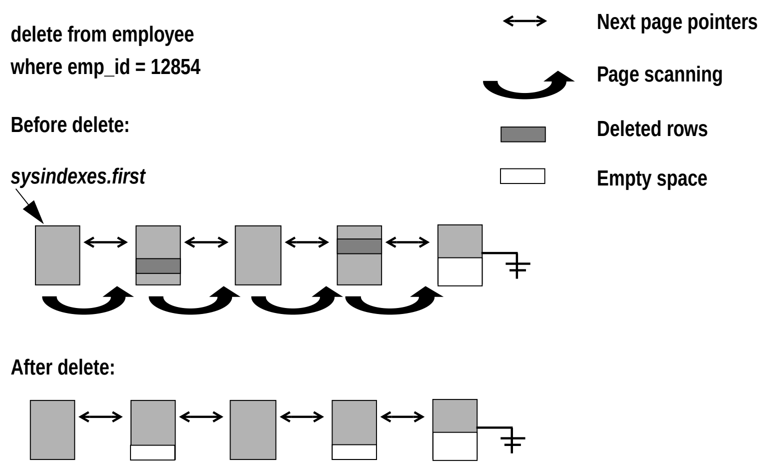


Figure 3-10: Deleting rows from an allpages-locked heap table

Deleting from a Data-Only Locked Heap Table

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

The space on the page is not recovered immediately. Rows in data-only-locked tables must maintain fixed row IDs, and need to be re-inserted in the same place if the transaction is rolled back, as shown in Figure 3-11.

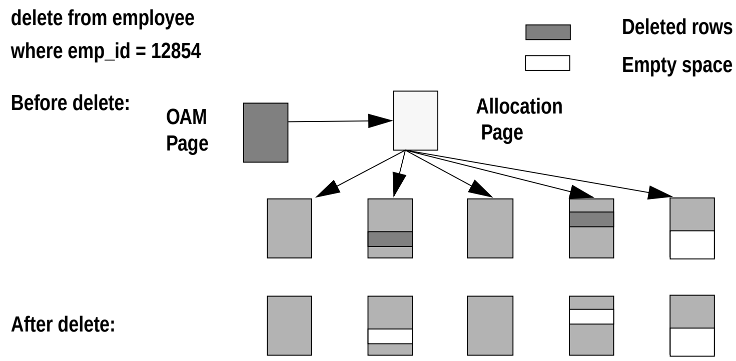


Figure 3-11: Deleting rows from a data-only-locked heap table

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

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

Deleting the Last Row on a Page

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

Updates on allpages-locked heap tables can be performed in several ways:

- If the length of the row does not change, the updated row replaces the existing row, and no data moves on the page.
- If the length of the row changes, and there is enough free space on the page, the row remains in the same place on the page, but other rows move up or down to keep the rows contiguous on the page. The row offset pointers at the end of the page are adjusted to point to the changed row locations.
- If the row does not fit on the page, the row is deleted from its current page, and the "new" row is inserted on the last page of the table. This type of update can cause contention on the last page of the heap, just as inserts do. If there are any nonclustered indexes on the table, all index references to the row need to be updated.

Updates on Data-Only-Locked Heap Tables

One of the requirements for data-only-locked tables is that the row ID of a data row never changes (except for intentional rebuilds of the table). Therefore, updates to data-only-locked tables can be performed by the first two methods described above, as long as the row fits on the page. But when a row in a data-only-locked table is updated so that it no longer fits on the page, a process called **row forwarding** performs the following steps:

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

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

If the row needs to be forwarded a second time, the original location is updated to point to the new page—the forwarded row is never more than one hop away from its original location. Figure 3-12 shows an update that causes row forwarding.

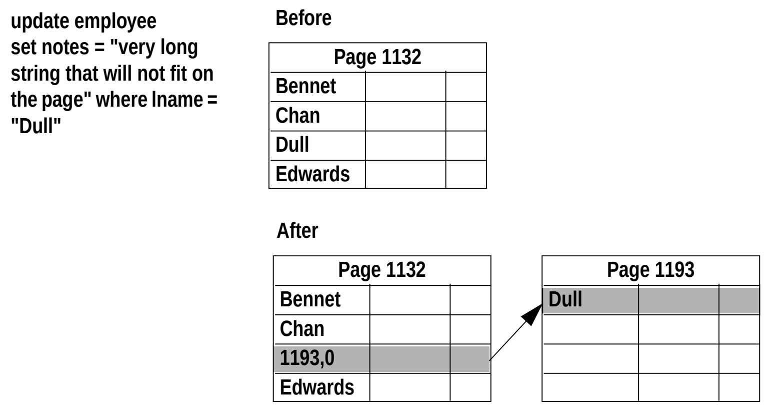


Figure 3-12: Original row stores pointer to the forwarded row's location

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

Forwarded rows can be cleared from a table using the **reorg** command. For more information on updates, see "How Update Operations Are Performed" on page 6-30.

How Adaptive Server Performs I/O for Heap Operations

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

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

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

- Choose to prefetch up to eight data pages at a time

- Choose between different caching strategies

Sequential Prefetch, or Large I/O

Adaptive Server's data caches can be configured by a System Administrator to allow large I/Os. When a cache is configured to allow large I/Os, Adaptive Server can choose to prefetch data pages.

Caches 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. Figure 3-13 shows large I/Os aligned within an extent.

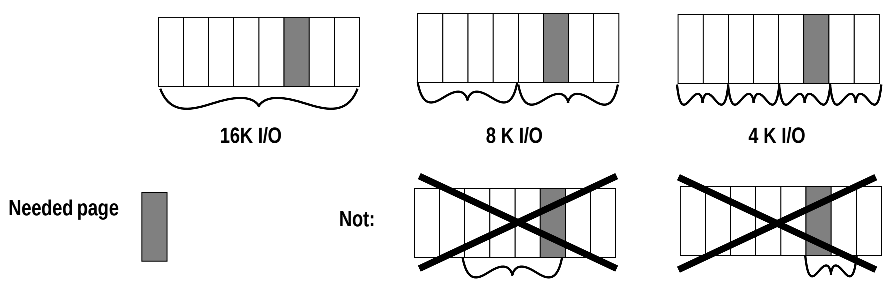


Figure 3-13: Alignment when performing large I/O

Since much of the time required to perform I/O operations is taken up in seeking and positioning, reading eight pages in a 16K I/O performs nearly eight times as fast as a single-page, 2K I/O, so queries that table scan should perform much better using large I/O. When several pages are read into cache with a single I/O, they are treated as a unit: they age in cache together, and if any page in the unit has been changed while the buffer was in cache, all pages are written to disk as a unit.

For more information on configuring memory caches for large I/O, See Chapter 32, "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 data cache. The default data cache can be configured for large I/O. If your applications include some heap tables, they will probably perform best when they use a cache configured for 16K I/O.

Heaps, I/O, and Cache Strategies

Each Adaptive Server data cache is managed as an MRU/LRU (most recently used/least recently used) chain of buffers. As buffers age in the cache, they move from the MRU end toward the LRU end. When 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 any pages that changed while they were in cache. This helps ensure that when the pages reach the LRU end of the cache, they are clean and can be reused.

Overview of Cache Strategies

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

- LRU replacement strategy, usually used for pages that a query needs to access more than once or pages that must be updated
- MRU, or **fetch-and-discard** replacement strategy, used for pages that a query needs to read only once

LRU Replacement Strategy

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

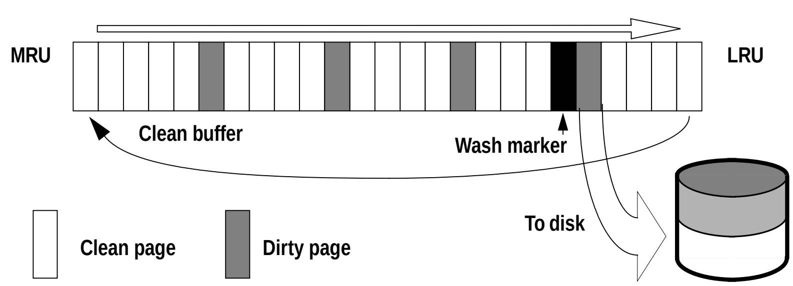


Figure 3-14: LRU strategy takes a clean page from the LRU end of the cache

When LRU Strategy Is Used

Adaptive Server uses LRU strategy for:

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

MRU Replacement Strategy

MRU (fetch-and-discard) replacement strategy is used for table scanning on heaps. This strategy places pages into the cache just before the wash marker, as shown in Figure 3-15.

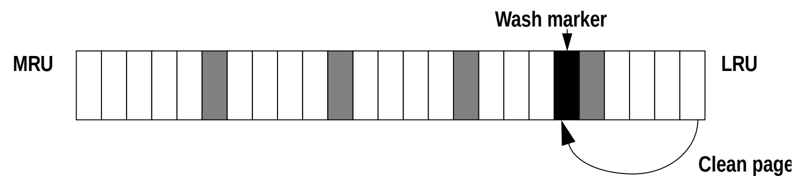


Figure 3-15: MRU strategy places pages just before the wash marker

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

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

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

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

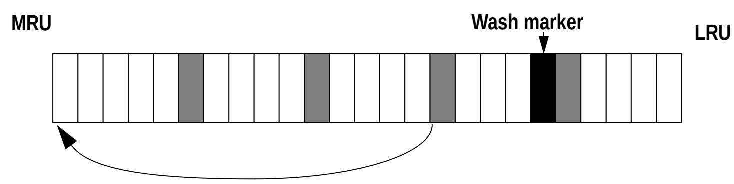


Figure 3-16: Finding a needed page in cache

Select Operations and Caching

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

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

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

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

➤ ***Note***

Large I/O on allpages-locked heaps is effective only when the page chains are not fragmented. See “Maintaining Heaps” on page 3-26 for information on maintaining heaps.

Data Modification and Caching

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

Caching and Inserts on Heaps

For inserts to heap tables, the insert takes place:

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

If an insert is the first row on a new page for the table, a clean data buffer is allocated to store the data page, as shown in Figure 3-17. 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.

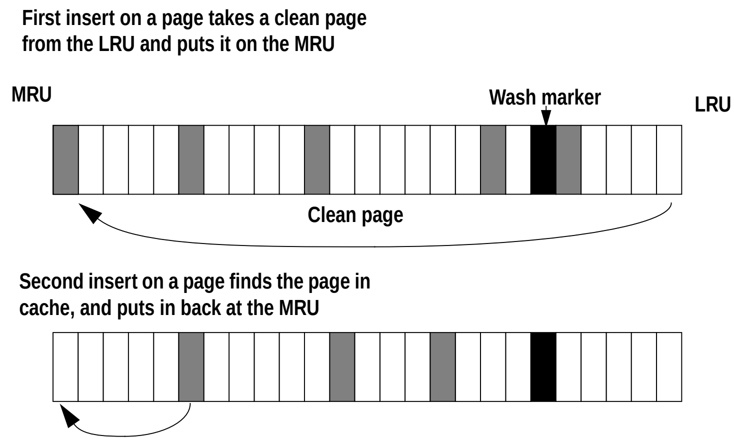


Figure 3-17: Inserts to a heap page in the data cache

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

- The page moves past the wash marker.
- A checkpoint or the housekeeper task writes it to disk. "The Data Cache" on page 32-7 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 row is changed and then the whole buffer (a single page, or more, depending on the I/O size) is placed on the MRU end of the chain. If the page is not in cache, it is read from disk into cache and examined to determine whether the rows on the page match query clauses. Its placement on the MRU/LRU chain depends on whether data on the page needs to be changed:

- If data on the page needs to be changed, the buffer is placed on the MRU end. It remains in cache, where it can be updated repeatedly or read by other users before being flushed to disk.

- If data on the page does not need to be changed, the buffer is placed just before the wash marker in the cache.

Asynchronous Prefetch and I/O on Heap Tables

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

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

Type of I/O	Loops	Steps in Each Loop
2K I/O no prefetch	1000	Request a page. Sleep until the page has been read from disk. Wait for a turn to run on the ASE engine (CPU). Read the rows on the page.
16K I/O no prefetch	125	Request an extent. Sleep until the extent has been read from disk. Wait for a turn to run on the ASE engine (CPU). Read the rows on the 8 pages.
Prefetch	4	Request all the pages in an allocation unit. Sleep until the first page has been read from disk. Wait for a turn to run on the ASE engine (CPU). Read all the rows on all the pages in cache.

Actual performance depends on cache size and other activity in the data cache. For more information on asynchronous prefetching, see Chapter 34, "Tuning Asynchronous Prefetch."

Heaps: Pros and Cons

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

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

Heaps work well for small tables and tables where changes are infrequent, but they do not work well for most large tables for queries that need to return a subset of the rows.

Heaps can be useful for tables that:

- Are fairly small and use only a few pages
- Do not require direct access to a single, random row
- Do not require ordering of result sets

Partitioned heaps are useful for tables with frequent, large volumes of batch inserts where the overhead of dropping and creating clustered indexes is unacceptable. With this exception, there are very few justifications for heap tables. Most applications perform better with clustered indexes on the tables.

Maintaining Heaps

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

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

Methods for Maintaining Heaps

After deletes and updates have left empty space on pages or have left empty pages on extents, use one of the following techniques to reclaim space in heap tables:

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

Using *reorg rebuild* to Reclaim Space

reorg rebuild copies all data rows to new pages and rebuilds any nonclustered indexes on the heap table. **reorg rebuild** can be used only on data-only-locked tables.

Reclaiming Space by Creating a Clustered Index

You can create and drop a clustered index on a heap table to reclaim space if updates and deletes have created many partially full pages in the table. To create a clustered index, you must have free space in the database of at least 120% of the table size. See "Determining the Space Available for Maintenance Activities" on page 30-10 for more information.

Reclaiming Space Using *bcp*

To reclaim space with **bcp**:

1. Copy the table out to a file using **bcp**.
2. Truncate the table with the **truncate table** command.
3. Copy the table back in again with **bcp**.

See "Steps for Partitioning Tables" on page 33-32 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. The transaction log does not have any indexes.

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 21, "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.

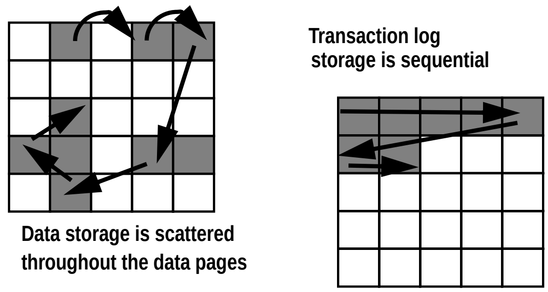


Figure 3-18: Data vs. log I/O

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

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).

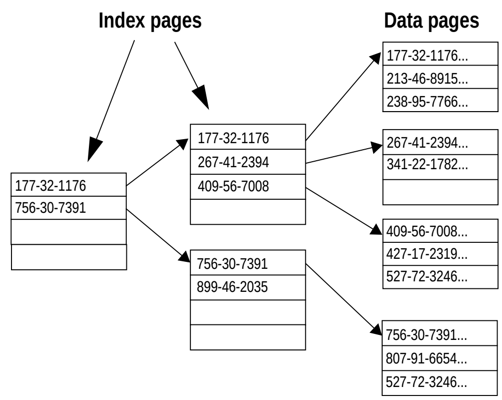


Figure 4-1: A simplified index schematic

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:
  - For allpages-locked tables, rows are stored in key order on pages, and pages are linked in key order.
  - For data-only-locked tables, indexes are used to direct the storage of data on rows and pages, but strict key ordering is not maintained.
- Nonclustered indexes, where the storage order of data in the table is not related to index keys

You can 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, or to individual data rows. Adaptive Server uses B-tree indexing, so each node in the index structure can have multiple children.

Index entries are usually much smaller than a data row in a data page, and index pages are much more densely populated than data pages. If a data row has 200 bytes (including row overhead), there are 10 rows per page. An index on a 15-byte field has about 100 rows per index page (the pointers require 4–9 bytes per row, depending on the type of index and the index level).

Indexes can have multiple levels:

- Root level
- Leaf level
- Intermediate level

Root Level

The root level is the highest level of the index. There is only one root page. If an allpages-locked table is very small, so that the entire index fits on a single page, there are no intermediate or leaf levels, and the root page stores pointers to the data pages. Data-only-locked tables always have a leaf level between the root page and the data pages.

For larger tables, the root page stores pointers to the intermediate level index pages or to leaf-level pages.

.

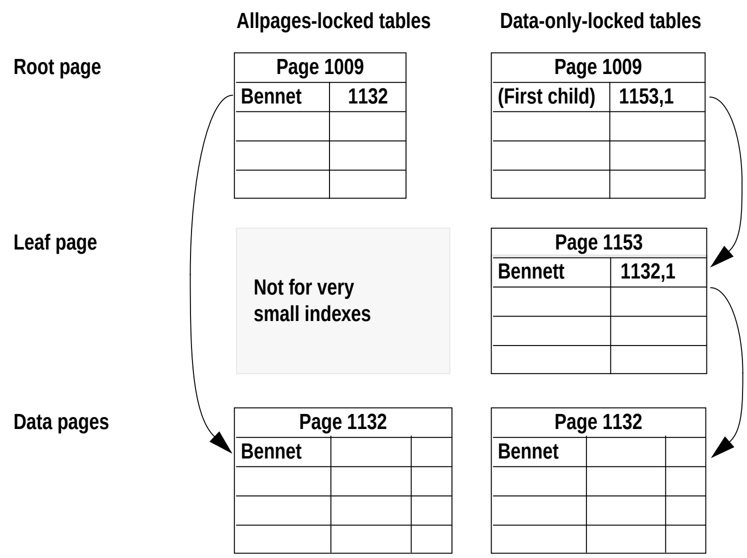


Figure 4-2: Data-only-locked tables always contain root and leaf levels

Leaf Level

The lowest level of the index is the leaf level. At the leaf level, the index contains a key value for each row in the table, and the rows are stored in sorted order by the index key:

- For clustered indexes on allpages-locked tables, the leaf level is the data. No other level of the index contains one index row for each data row.
- For nonclustered indexes and clustered indexes on data-only-locked tables, the leaf level contains the index key value for each row, a pointer to the page where the row is stored, and a pointer to the rows on the data page. The leaf level is the level just above the data; it contains one index row for each data row. Index rows on the index page are stored in key value order.

Intermediate Level

All levels between the root and leaf levels are intermediate levels. An index on a large table or an index using long keys may have many intermediate levels. A very small allpages-locked table may not have an intermediate level at all; the root pages point directly to the leaf level.

Clustered Indexes on Allpages-Locked Tables

In clustered indexes on allpages-locked tables, leaf-level pages are also the data pages, and all rows are kept in physical order by the keys. Clustered indexes on allpages-locked tables are structurally different than nonclustered indexes.

➤ ***Note***

For data-only-locked tables, clustered indexes are structured like nonclustered indexes. For information on clustered indexes on data-only-locked tables, see “Clustered Indexes on Data-Only-Locked Tables” on page 4-20.

In an allpages-locked table with a clustered index, data rows are physically ordered by the index key. Physical ordering means that:

- All entries on a data page are in index key order.
- By following the “next page” pointers on the data pages, Adaptive Server reads the entire table in index key order.

On the root and intermediate pages, each entry points to a page on the next level. Figure 4-3 shows a clustered index.

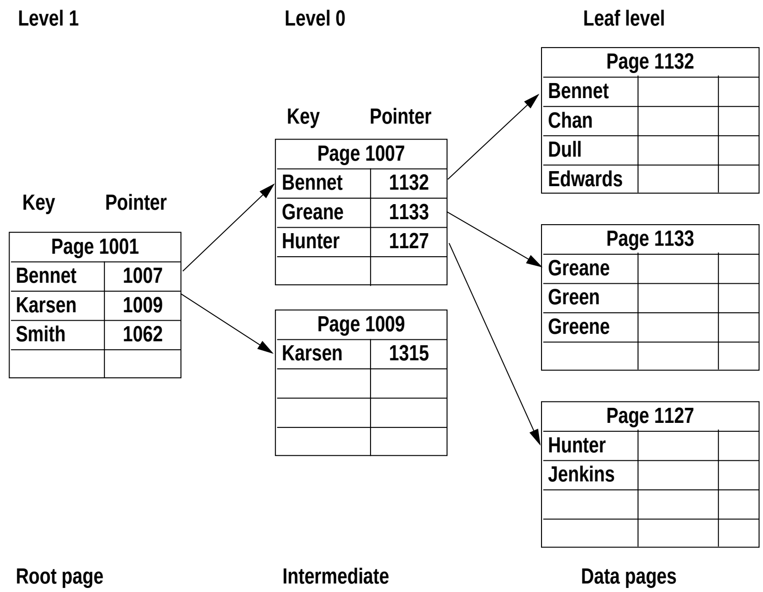


Figure 4-3: Clustered index on last name, on an allpages-locked table

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.

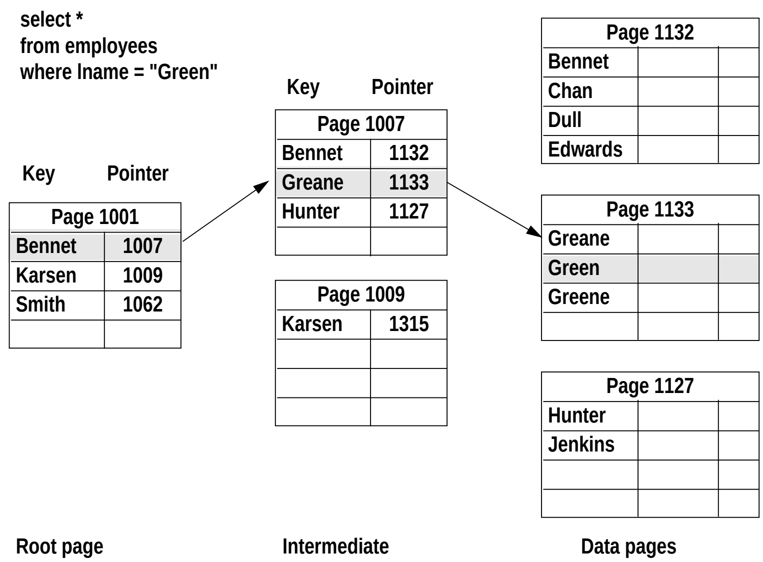


Figure 4-4: Selecting a row using a clustered index, allpages-locked table

On the root level page, “Green” is greater than “Bennet,” but less than Karsen, so the pointer for “Bennet” is followed to page 1007. On page 1007, “Green” is greater than “Greane,” but less than “Hunter,” so the pointer to page 1133 is followed to the data page, where the row is located and returned to the user.

This retrieval via the clustered index requires:

- One read for the root level of the index
- One read for the intermediate level
- One read for the data page

These reads may come either from cache (called a **logical read**) or from disk (called a **physical read**). On tables that are frequently used, the higher levels of the indexes are often found in cache, with lower levels and data pages being read from disk.

Clustered Indexes and Insert Operations

When you insert a row into an allpages-locked table with a clustered index, the data row must be placed in physical order according to the key value on the table. Other rows on the data page move down on the page, as needed, to make room for the new value. As long as there is room for the new row on the page, the insert does not affect any other pages in the database. The clustered index is used to find the location for the new row.

Figure 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.

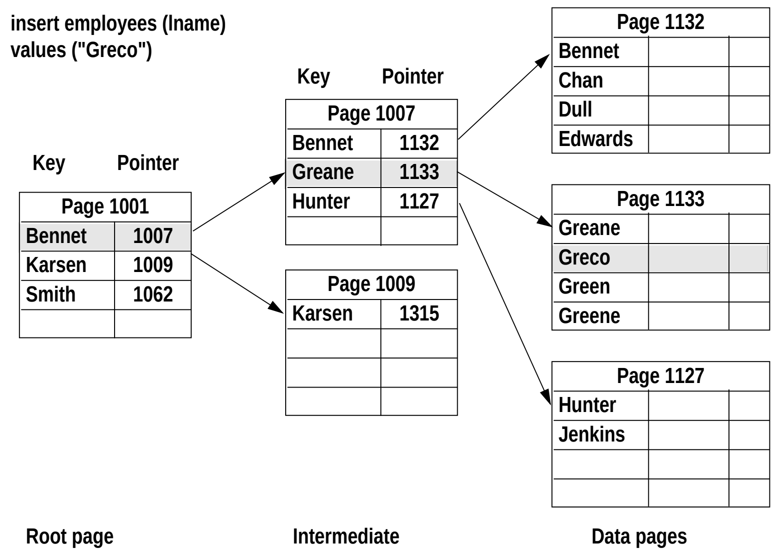


Figure 4-5: Inserting a row into an allpages-locked table with a clustered index

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 available, 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 pointers to the affected data rows must be changed to point to the new page and row locations.

In some cases, page splitting is handled slightly differently. See "Exceptions to Page Splitting" on page 4-9.

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

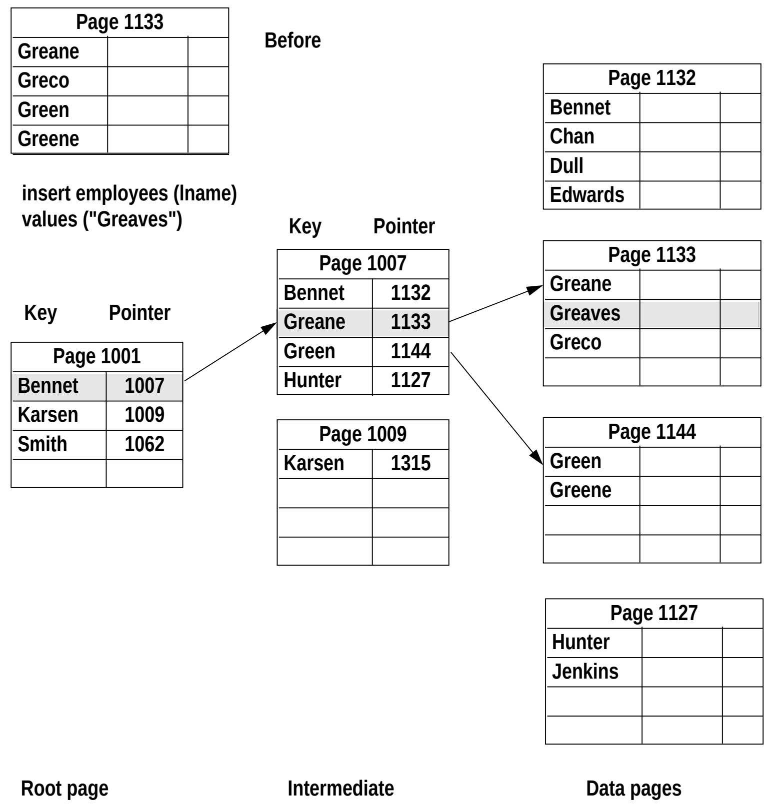


Figure 4-6: Page splitting in an allpages-locked table with a clustered index

Exceptions to Page Splitting

There are exceptions to 50-50 page splits:

- If you insert a huge row that cannot fit on either the page before or the page after the page that requires splitting, two new pages are allocated, one for the huge row and one for the rows that follow it.

- If possible, Adaptive Server keeps duplicate values together when it splits pages.
- If Adaptive Server detects that all inserts are taking place at the end of the page, due to a increasing key value, the page is not split when it is time to insert a new row that does not fit at the bottom of the page. Instead, a new page is allocated, and the row is placed on the new page.
- If Adaptive Server detects that inserts are taking place in order at other locations on the page, the page is split at the insertion point.

Page Splitting on Index Pages

If a new row needs to be added to a full index page, the page split process on the index page is similar to the data page split. A new page is allocated, and half of the index rows are moved to the new page. A new row is inserted at the next highest level of the index to point to the new index page.

Performance Impacts of Page Splitting

Page splits are expensive operations. In addition to the actual work of moving rows, allocating pages, and logging the operations, the cost is increased by:

- Updating the clustered index itself
- Updating the page pointers on adjacent pages to maintain page linkage
- Updating all nonclustered index entries that point to the rows affected by the split

When you create a clustered index for a table that will grow over time, you may want to use **fillfactor** to leave room on data pages and index pages. This reduces the number of page splits for a time. See "Choosing Space Management Properties for Indexes" on page 9-19.

Overflow Pages

Special overflow pages are created for nonunique clustered indexes on allpages-locked tables when a newly inserted row has the same key as the last row on a full data page. A new data page is allocated and linked into the page chain, and the newly inserted row is placed on the new page (see Figure 4-7).

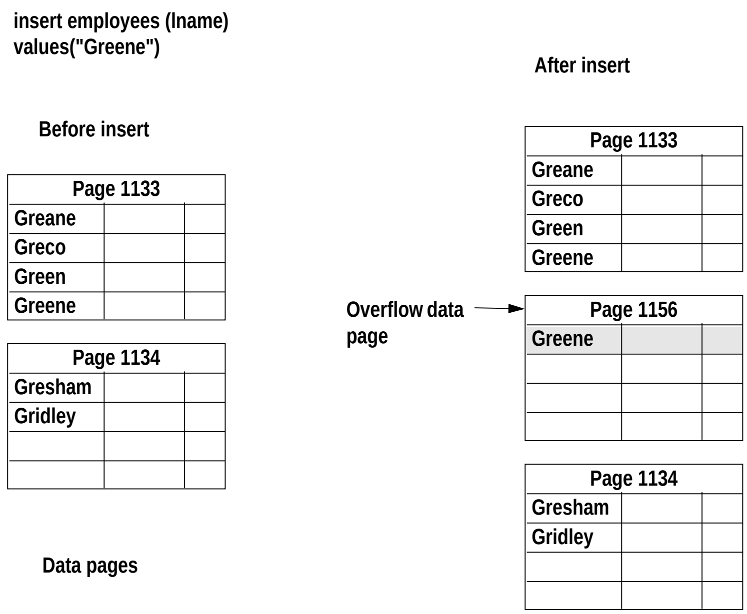


Figure 4-7: Adding an overflow page to a clustered index, allpages-locked table

The only rows that will be placed on this overflow page are additional rows with the same key value. In a nonunique clustered index with many duplicate key values, there can be numerous overflow pages for the same value.

The clustered index does not contain pointers directly to overflow pages. Instead, the next page pointers are used to follow the chain of overflow pages until a value is found that does not match the search value.

Clustered Indexes and Delete Operations

When you delete a row from an allpages-locked table that has a clustered index, other rows on the page 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.

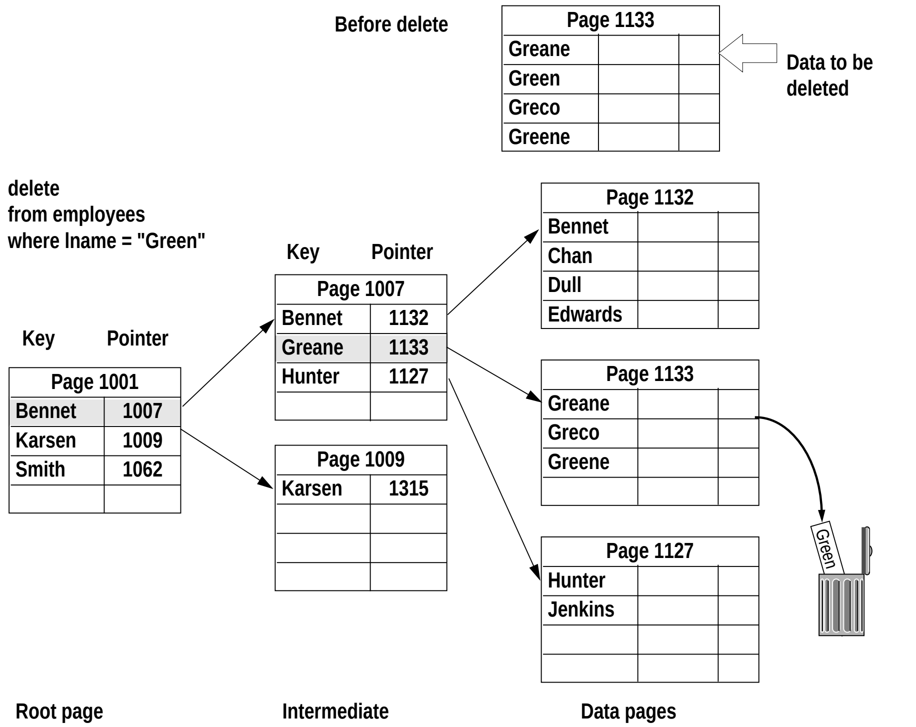


Figure 4-8: Deleting a row from a table with a clustered index

Deleting the Last Row on a Page

If you delete the last row on a data page, the page is deallocated and the next and previous page pointers on the adjacent pages are changed. The rows that point to that page in the leaf and intermediate levels of the index are removed.

If the deallocated data page is on the same extent as other pages belonging to the table, it can be used again when that table needs an additional page. If the deallocated data page is the last page on the extent that belongs to the table, the extent is also deallocated and becomes available for the expansion of other objects in the database.

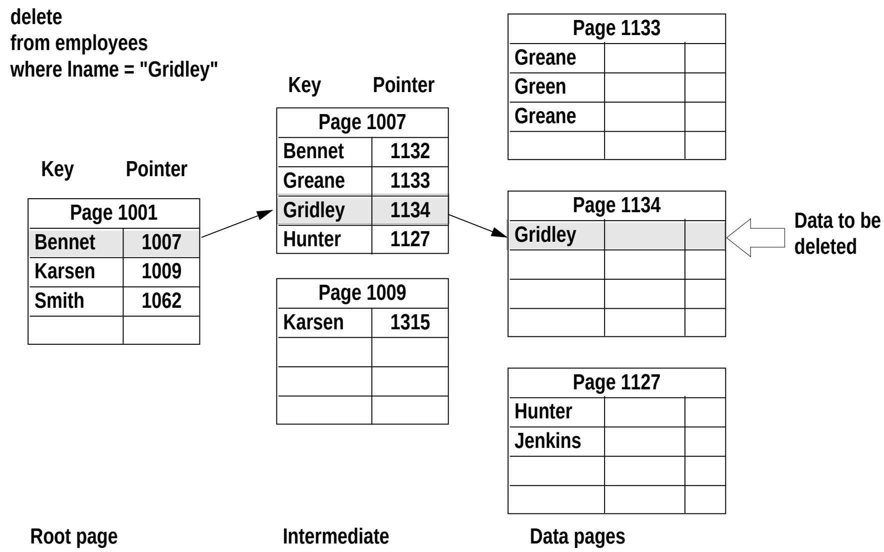


Figure 4-9: Deleting the last row on a page (before the delete)

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 used space contiguous.

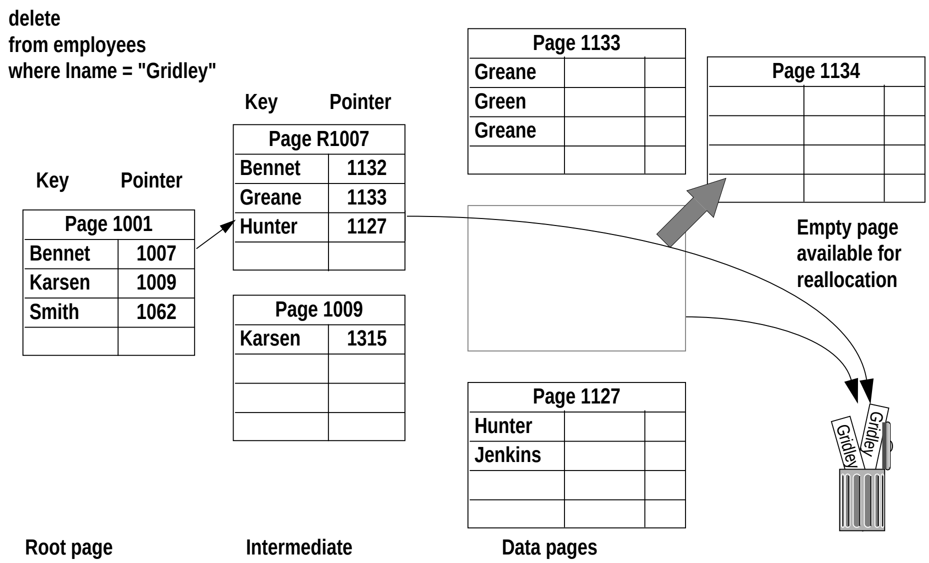


Figure 4-10: Deleting the last row on a page (after the delete)

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 indexes on allpages-locked tables and other 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 on allpages-locked tables, the data rows are stored in order by the index keys, so by definition, the data level is the leaf level. There is no other level of the clustered index that contains one index row for each data row. Clustered indexes on allpages-locked tables are sparse indexes. The level above the data contains one pointer for every data **page**, not data **row**.

In nonclustered indexes and clustered indexes on data-only-locked tables, the level just above the data is the leaf level: it contains a key-pointer pair for each data row. These indexes are dense. At the level above the data, they contain one index row for each data row.

Nonclustered Index Structure

The table in Figure 4-11 shows a nonclustered index on *lname*. The data rows at the far right show pages in ascending order by *employee_id* (10, 11, 12, and so on) because there is a clustered index on that column.

The root and intermediate pages store:

- The key value
- The row ID
- The pointer to the next level of the index

The leaf level stores:

- The key value
- The row ID

The row ID in higher levels of the index is used for indexes that allow duplicate keys. If a data modification changes the index key or deletes a row, the row ID positively identifies all occurrences of the key at all index levels.

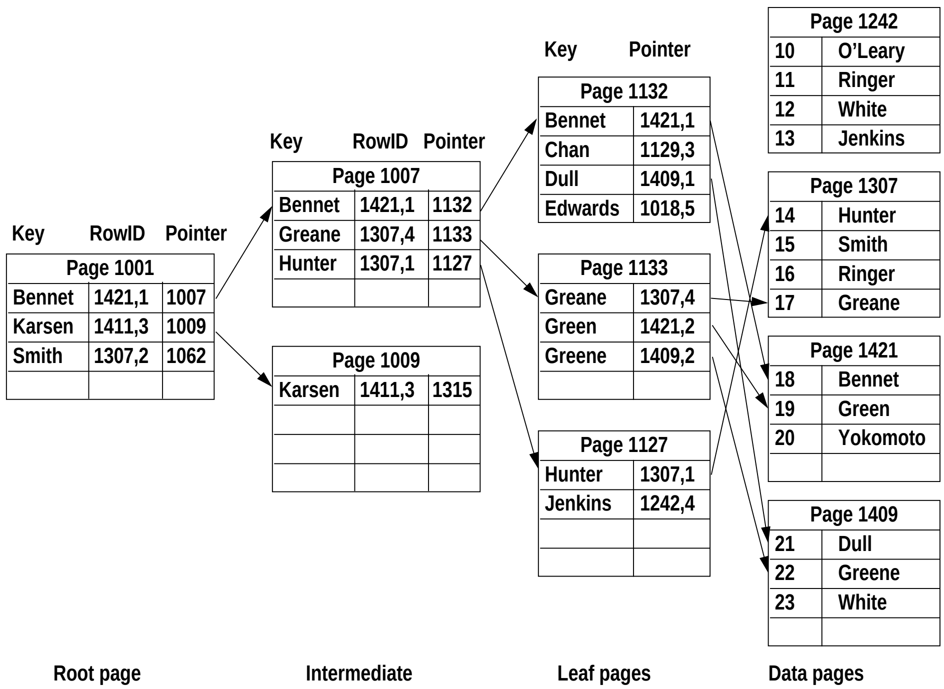


Figure 4-11: Nonclustered index structure

Nonclustered Indexes and Select Operations

When you select a row using a nonclustered index, the search starts at the root level. *sysindexes.root* stores the page number for the root page of the nonclustered index.

In Figure 4-12, "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 row 2 on page 1421. This page is fetched, the "2" byte in the offset table is checked, and the row is returned from the byte position on the data page.

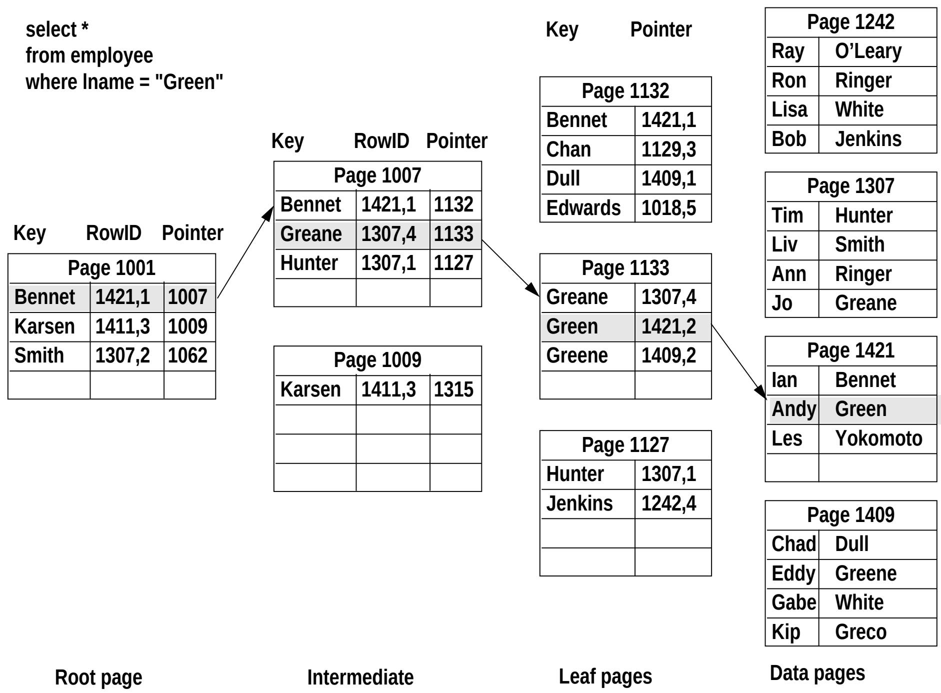


Figure 4-12: Selecting rows using a nonclustered index

Nonclustered Index Performance

The query in Figure 4-12 requires the following I/O:

- One read for the root level page
- One read for the intermediate level page
- One read for the leaf-level page
- One read for the data page

If your applications use a particular nonclustered index frequently, the root and intermediate pages will probably be in cache, so only one or two physical disk I/Os need to be performed.

Nonclustered Indexes and Insert Operations

When you insert rows into a heap that has a nonclustered index and no clustered index, the insert goes to the last page of the table. If the heap is partitioned, the insert goes to the last page on one of the partitions. Then, the nonclustered index is updated to include the new row. If the table has a clustered index, it is used to find the location for the row. The clustered index is updated, if necessary, and each nonclustered index is updated to include the new row.

Figure 4-13 shows an insert into a heap table with a nonclustered index. The row is placed at the end of the table. A row containing the new key value and the row ID is also inserted into the leaf level of the nonclustered index.

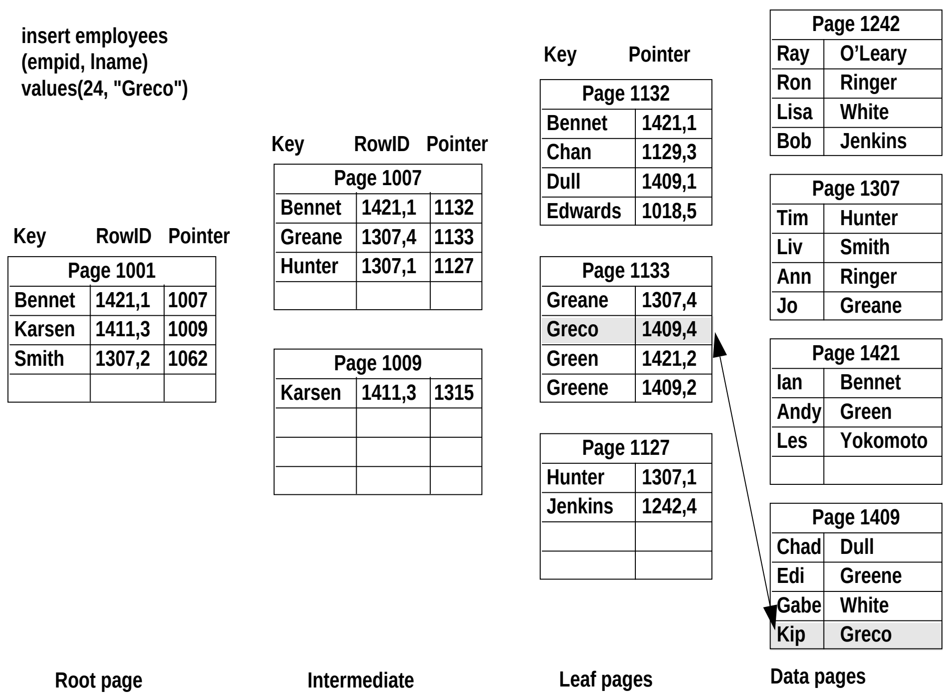


Figure 4-13: An insert into a heap table with a nonclustered index

Nonclustered Indexes and Delete Operations

When you delete a row from a table, the query can use a nonclustered index on the column or columns in the **where** clause to locate the data row to delete, as shown in Figure 4-14. 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.

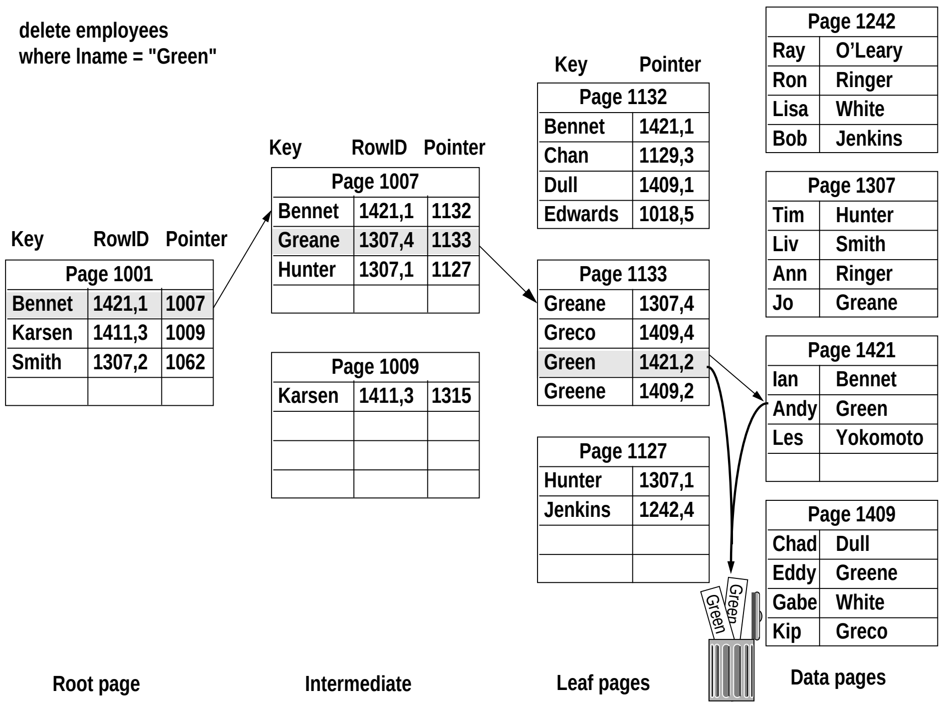


Figure 4-14: Deleting a row from a table with a nonclustered index

If the delete operation removes the last row on the data page, the page is deallocated and the adjacent page pointers are adjusted in allpages-locked tables. Any references to the page are also deleted in higher levels of the index.

If the delete operation leaves only a single row on an index intermediate page, index pages may be merged, as with clustered indexes. See "Index Page Merges" on page 4-14.

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

Clustered Indexes on Data-Only-Locked Tables

Clustered indexes on data-only-locked tables are structured like nonclustered indexes. They have a leaf level above the data pages. The leaf level contains the key values and row ID for each row in the table.

Unlike clustered indexes on allpages-locked tables, the data rows in a data-only-locked table are not necessarily maintained in exact order by the key. Instead, the index directs the placement of rows to pages that have adjacent or nearby keys. When a row needs to be inserted in a data-only-locked table with a clustered index, the insert uses the clustered index key just before the value to be inserted. The index pointers are used to find that page, and the row is inserted on the page if there is room. If there is not room, the row is inserted on a page in the same allocation unit, or on another allocation unit already used by the table.

To provide nearby space for maintaining data clustering during inserts and updates to data-only-locked tables, you can set space management properties to provide space on pages (using **fillfactor** and **exp_row_size**) or on allocation units (using **reservepagegap**). See Chapter 31, “Setting Space Management Properties.”

Figure 4-15 shows a clustered index being used to direct an insert to a data page in a data-only-locked table.

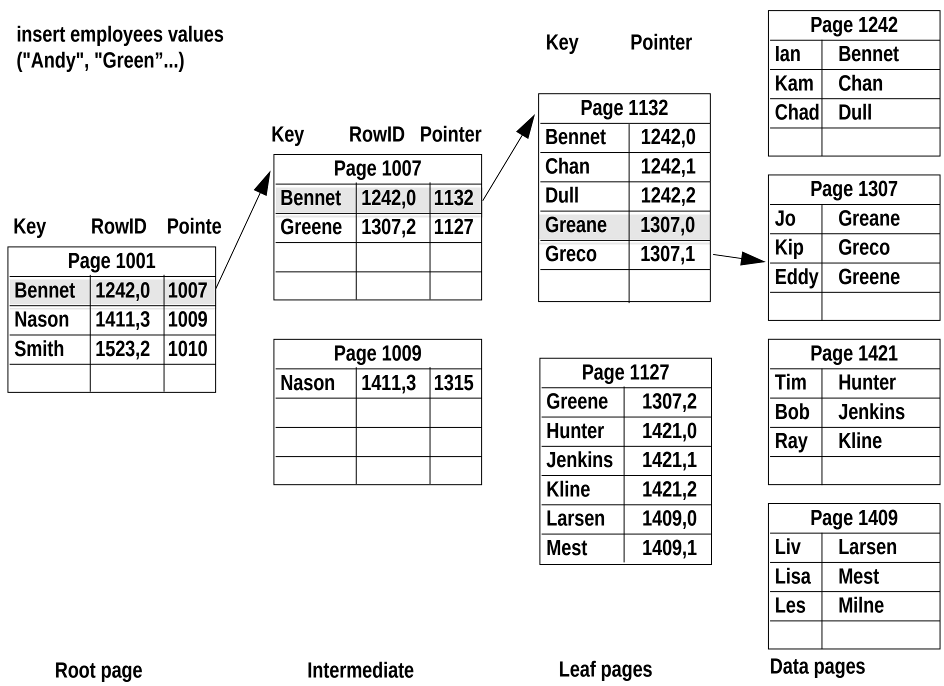


Figure 4-15: Insert using a clustered index on a data-only-locked table

Index Covering

Index covering is a indexing strategy that can produce dramatic performance improvements when all columns needed by the query are included in the index.

You can create indexes on more than one key. These are called **composite indexes**. Composite indexes can have up to 31 columns adding up to a maximum 600 bytes. If you create a composite nonclustered index on each column referenced in the query's select list and in any **where**, **having**, **group by**, and **order by** clauses, the query can be satisfied by accessing only the index. Since the leaf level of a nonclustered index or a clustered index on a data-only-locked table contains the key values for each row in a table, queries that access only the key values can retrieve the information by using the leaf level of the nonclustered index as if it were the actual table data. This is called index covering.

There are two types of index scans that can use an index that covers the query:

- The matching index scan
- The nonmatching index scan

For both types of covered queries, the index keys must contain all the columns named in the query. Matching scans have additional requirements. "Choosing Composite Indexes" on page 9-11 describes query types that make good use of covering indexes.

Covering 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.

For a covering matching index scan to be used, the index must contain all columns named in the query. Plus, 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 can be used only for nonmatching scans.

When doing a matching index scan, Adaptive Server uses standard index access methods to move from the root of the index to the nonclustered leaf page that contains the first row.

In Figure 4-16, the nonclustered index on *lname, fname* covers the query. The **where** clause includes the leading column, and all columns in the select list are included in the index, so the data page does not need to be accessed.

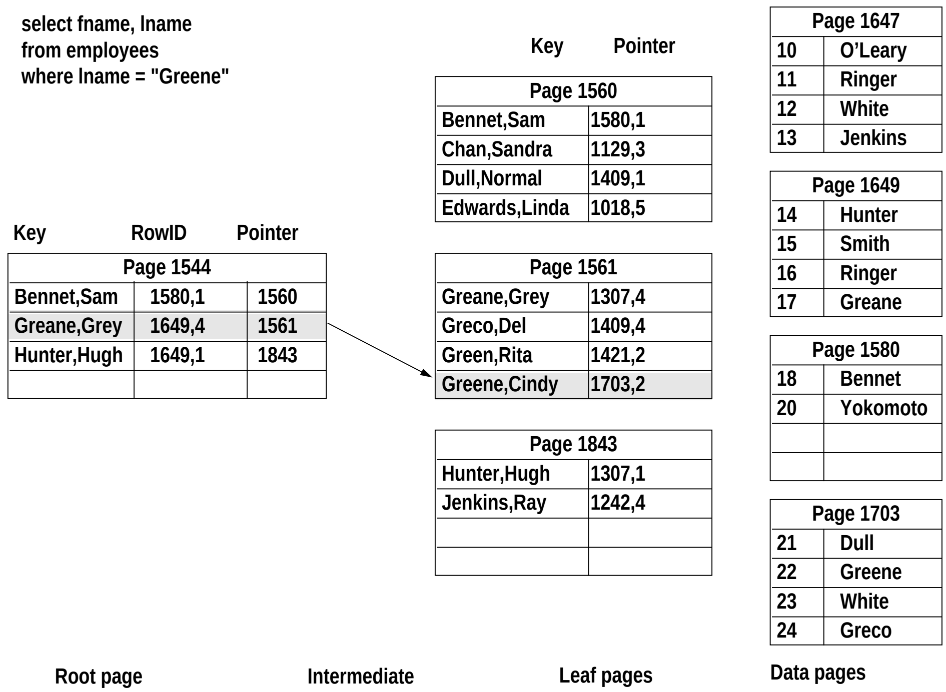


Figure 4-16: Matching index access does not have to read the data row

Covering Nonmatching Index Scans

When the columns specified in the **where** clause do not include the leading column in the index, but all columns named in the select list and other query clauses (such as **group by** or **having**) are included in the index, Adaptive Server saves I/O by scanning the entire leaf level of the index, rather than scanning the table. It cannot perform a matching scan because the first column of the index is not specified.

The query in Figure 4-17 shows a nonmatching index scan. This query does not use the leading columns on the index, but all columns required in the query are in the nonclustered index on *lname, fname, emp_id*. The nonmatching scan must examine all rows on the leaf level. It scans all leaf level index pages, starting from the first page. It has no way of knowing how many rows might match the query conditions, so it must examine every row in the index. Since it must begin at the first page of the leaf level, it can use the pointer in *sysindexes.first* rather than descending the index.

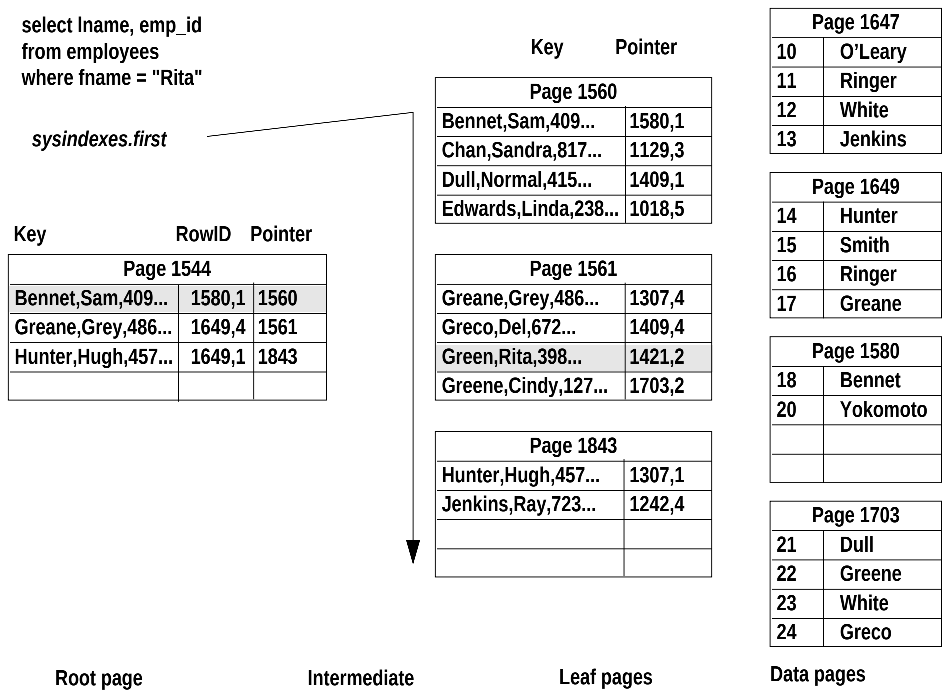


Figure 4-17: A nonmatching index scan

Indexes and Caching

"How Adaptive Server Performs I/O for Heap Operations" on page 3-19 introduces the basic concepts of the Adaptive Server data cache, and shows how caches are used when reading heap tables. Index pages get special handling in the data cache, as follows:

- Root and intermediate index pages always use LRU strategy.
- Index pages can use one cache while the data pages use a different cache, if the index is bound to a different cache.
- Covering index scans can use fetch-and-discard strategy.
- Index pages can cycle through the cache many times, if **number of index trips** is configured.

When a query that uses an index is executed, the root, intermediate, leaf, and data pages are read in that order. If these pages are not in cache, they are read into the MRU end of the cache and are moved toward the LRU end as additional pages are read in. Figure 4-18 shows the data cache just after these 4 pages have been read.

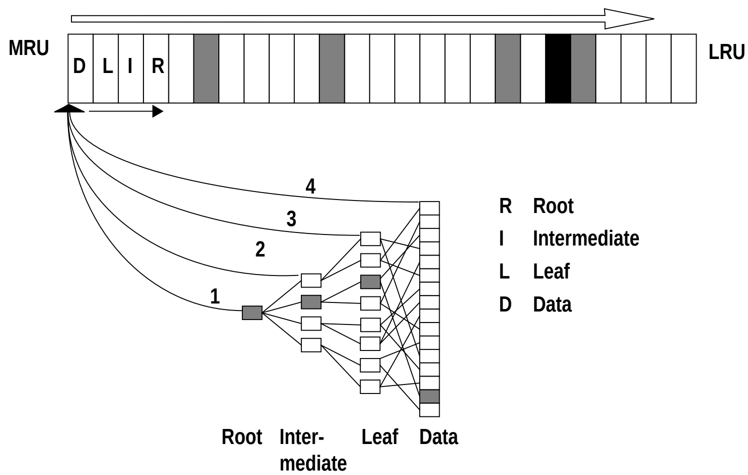


Figure 4-18: Caching used for a point query via a nonclustered index

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 4-19 shows a root page being moved back to the top of the cache for a second query that uses the same index, accessing different intermediate, leaf, and data pages.

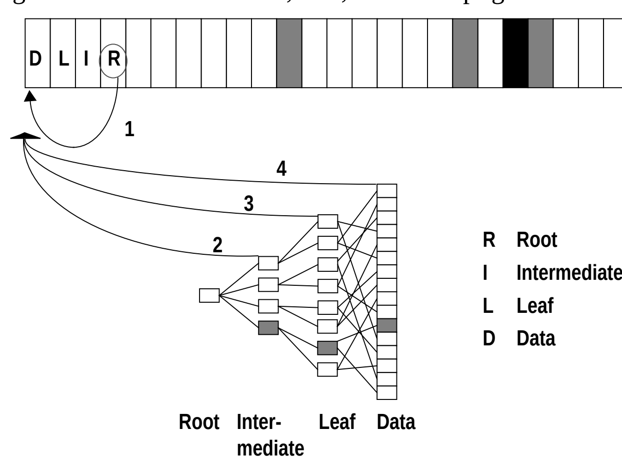


Figure 4-19: Finding the root index page in cache

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 4-20 shows index pages being read into one cache and a data page into a different cache.

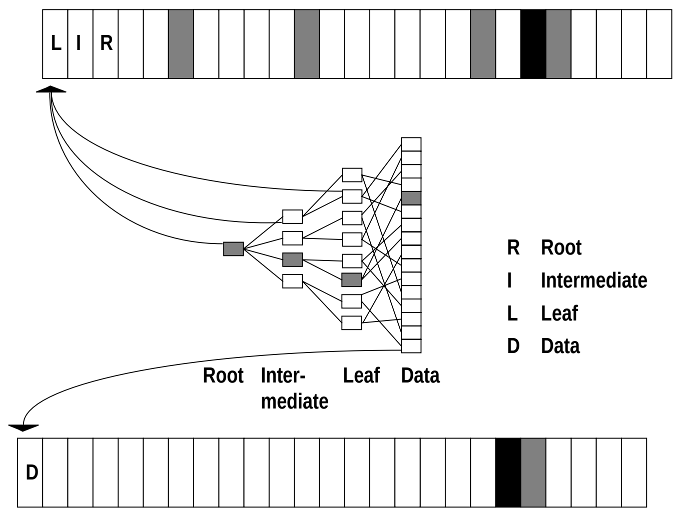


Figure 4-20: Caching with separate caches for data and index pages

Index Trips Through the Cache

A special strategy keeps index pages in cache. Data pages make only a single trip through the cache: they are read in at the MRU end of the cache or placed just before the wash marker, depending on the cache strategy chosen for the query. Once the pages reach the LRU end of the cache, the buffer for that page is reused when another page needs to be read into cache.

For index pages, a counter controls the number of trips that an index page can make through the cache. When the counter is greater than 0 for an index page, and it reaches the LRU end of the page chain, the counter is decremented by 1, and the page is placed at the MRU end again. Figure 4-21 shows an index page moving from the LRU end of the cache to the MRU end.

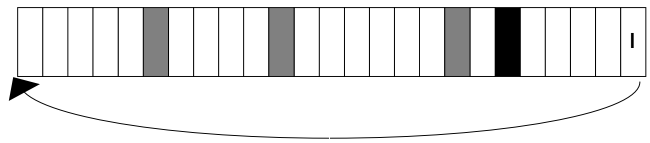


Figure 4-21: Index page recycling in the cache

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 17-30 of the *System Administration Guide*.

Tuning Query Performance

5 Understanding the Query Optimizer

This chapter introduces the Adaptive Server query optimizer and explains the steps performed when you run queries. It explains how statistics are used to analyze search arguments and join clauses.

This chapter contains the following sections:

This chapter explains how costs for individual query clauses are determined. Chapter 6, "Access Methods and Query Costing for Single Tables," explains how these costs are used to estimate the logical, physical, and total I/O cost for single table queries.

Chapter 7, "Access Methods and Query Costing for Joins and Subqueries," explains how costs are used when queries join two or more tables, or when queries include subqueries.

What Is Query Optimization?

Query optimization is the process of analyzing a query to determine what resources it requires and how to execute the query with the least possible query cost. To understand the optimization of a query, you need to understand how the query 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. Chapters 3 and 4 provide a basic introduction to how tables and indexes are stored and accessed.

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 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.
- where clauses are written in unoptimizable form.
- 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 and statistics about the tables, indexes, and columns named in the 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 **access methods** (table scan or index scan, type of join to use, join order, and so on) to access each table. A query plan is compiled code that is ready to run.

Adaptive Server's query optimizer models Adaptive Server's access methods. Using statistics on tables and indexes, it predicts the cost of using alternative access methods to resolve a particular query. The Adaptive Server optimizer finds the best query plan—the plan that is least the costly in terms of I/O. For many queries, there are many possible query plans. The optimizer uses statistics on tables and indexes to estimate the cost of the available access methods for the query. Adaptive Server selects the least costly plan, and compiles and executes it.Figure 5-1 shows query processing steps and optimizer inputs.

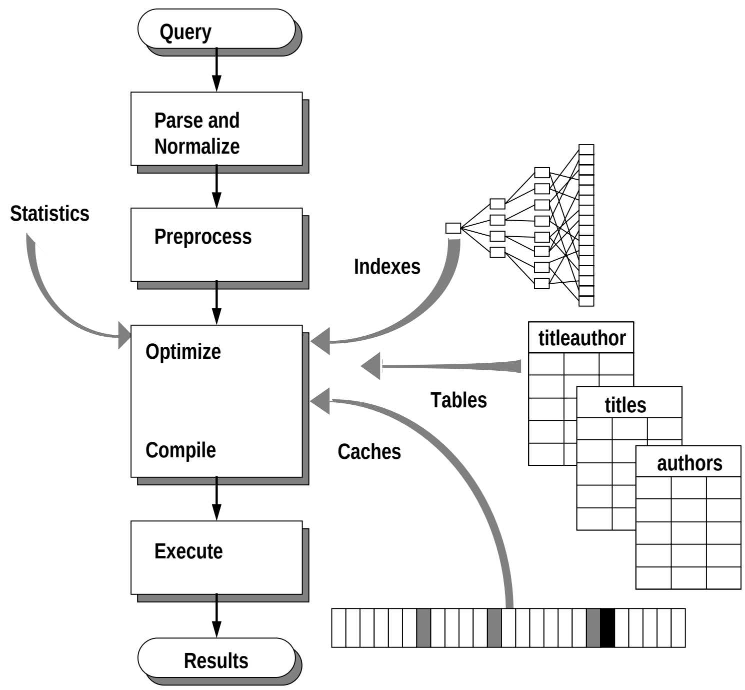


Figure 5-1: Query execution steps

Steps in Query Processing

When you execute a 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. Permissions are checked to ensure

that the user has permission to access all tables and columns in the query.

2. Preprocessing changes some search arguments to optimizable form and adds optimizable search arguments and join clauses.
3. 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.
   - The cost of using each index that matches a search argument or join column is estimated.
   - The join order and join type are chosen.
   - The final access method is determined.
4. The chosen query plan is compiled.
5. 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 for each table that reduces the total time needed to process a query. The optimizer bases its choice on the statistics available for the tables being queried and on other factors such as cache strategies, cache size, and I/O size. A major component of optimizer decision-making is the statistics available for the tables, indexes, and columns.

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 some situations, the optimizer may make the incorrect choice of access methods. In some cases, this may be the result of inaccurate or incomplete information (such as out-of-date statistics). 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.

How Is "Fast" Determined?

The query optimizer uses I/O cost as the measure of query execution cost. The significant costs in query processing are:

- Physical I/O, when pages must be read from disk

- Logical I/O, when pages in cache are read for a query

See “Basic Units of Costing” on page 6-3 for more information.

Factors Examined During Query Optimization

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, and the number of OAM and allocation pages that need to be read.
- The indexes that exist on the tables and columns used in the query, the type of index, and the height, number of leaf pages, and cluster ratios for each index.
- Whether the index covers the query, that is, whether the query can be satisfied by retrieving data from the index leaf pages 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.
- The size of the available data cache or caches, the size of I/O supported by the caches, and the cache strategy to be used.
- The cost of physical and logical reads.
- Optimizable join clauses and the best join order and join type, considering the costs and number of scans required for each join and the usefulness of indexes in limiting the I/O.
- 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 data or index 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 by computing the logical and physical I/Os. Adaptive Server then uses the cheapest plan.

Stored procedures and triggers are optimized when the object is first executed, and the query plan is stored in the procedure cache. If other users execute the same procedure while an unused copy of the plan resides in cache, the compiled query plan is copied in cache, rather than being recompiled.

How Preprocessing Adds Optimizable Clauses

After a query is parsed and normalized, but before the optimizer begins its analysis, the query is preprocessed to increase the number of optimizable clauses:

- Some search arguments are converted to equivalent arguments.
- Some expressions used as search arguments are preprocessed to generate a literal value that can be optimized.
- Search argument transitive closure is applied where possible.
- Join column transitive closure is applied where possible.
- For some queries that use **or**, additional search arguments can be generated to provide additional optimization paths.

The changes made by preprocessing are transparent unless you are examining the output of query tuning tools such as **showplan**, **statistics io**, or **dbcc traceon(302)**. If you run queries that benefit from the addition of optimizable search arguments, you see the added clauses:

- In additional costing blocks for the added optimizable clauses in **dbcc traceon(302)** output.
- In **showplan** output, you may see “Keys are” messages for tables where you did not specify a search argument or a join.

Converting Clauses to Search Argument Equivalents

Preprocessing looks for some query clauses that it can convert to the form used for search arguments (SARGs). These are listed in Table 5-1.

Table 5-1: Search argument equivalents

Clause	Conversion
between	Converted to **>=** and **<=** clauses. For example, **between 10 and 20** is converted to **>= 10 and <= 20**.
like	If the first character in the pattern is a constant, **like** clauses can be converted to greater than or less than queries. For example, **like "sm%"** becomes **>= "sm" and < "sn"**. If the first character is a wildcard, a clause such as **like "%x"** cannot use an index for access, but histogram values can be used to estimate the number of matching rows.
in (*values_list*)	Converted to a list of **or** queries, that is, **int_col in (1, 2, 3)** becomes **int_col = 1 or int_col = 2 or int_col = 3**

Conversion of Expressions into Search Arguments

Many expressions are converted into literal search strings before query optimization. In the following examples, the processed expressions are shown as they appear in the search argument analysis of **dbcc traceon(302)** output:

Operation	Example of where Clause	Processed Expression
Implicit conversion	`numeric_col = 5`	`numeric_col = 5.0`
Conversion function	`int_column = convert(int, "77")`	`int_column = 77`
Arithmetic	`salary = 5000*12`	`salary = 60000`
Math functions	`width = sqrt(900)`	`width = 30`
String functions	`shoe_width = replicate("E", 5)`	`shoe_width = "EEEEE"`
String concatenation	`full_name = "Fred" + " " + "Simpson"`	`full_name = "Fred Simpson"`
Date functions	`week = datepart(wk, "5/22/99")`	`week = 21` Note: **getdate()** is not optimizable.

These conversions allow the optimizer to use the histogram values for a column rather than using default selectivity values.

The following are exceptions:

- The **getdate** function
- Most system functions such as **object_id** or **object_name**

These are not converted to literal values before optimization.

Search Argument Transitive Closure

Preprocessing applies transitive closure to search arguments. For example, the following query joins *titles* and *titleauthor* on *title_id* and includes a search argument 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 search argument 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"
```

With this additional clause, the optimizer can use index statistics on *titles.title_id* to estimate the number of matching rows in the *titleauthor* table. The more accurate cost estimates improve index and join order selection.

Join Transitive Closure

Preprocessing applies transitive closure to join columns for normal equijoins if join transitive closure is enabled at the server or session level. The following query specifies the equijoin of *t1.c11* and *t2.c21*, and the equijoin of *t2.c21* and *t3.c31*:

```
select *
from t1, t2, t3
where t1.c11 = t2.c21
and t2.c21 = t3.c31
and t3.c31 = 1
```

Without join transitive closure, the only join orders considered are (*t1*, *t2*, *t3*), (*t2*, *t1*, *t3*), (*t2*, *t3*, *t1*),and (*t3*, *t2*, *t1*). By adding the join on *t1.c11* = *t3.31*, the optimizer expands the list of join orders with these possibilities: (*t1*, *t3*, *t2*) and (*t3*, *t1*, *t2*). Search argument transitive closure applies the condition specified by *t3.c31* = 1 to the join columns of *t1* and *t2*.

Transitive closure is used only for normal equijoins, as shown above. Join transitive closure is not performed for:

- Non-equijoins; for example, *t1.c1 > t2.c2*
- Equijoins that include an expression; for example, *t1.c1 = t2.c1 + 5*
- Equijoins under an **or** clause
- Outer joins; for example *t1.c11* *= *t2.c2* or **left join** or **right join**
- Joins across subquery boundaries
- Joins used to check referential integrity or the **with check option** on views
- Columns of incompatible datatypes

Enabling Join Transitive Closure

A System Administrator can enable join transitive closure at the server level with the configuration parameter **enable sort-merge joins and JTC**. This configuration parameter also enables merge joins. At the session level, **set jtc on** enables join transitive closure, and takes precedence over the server-wide setting. For more information on the types of queries likely to benefit from the use of join transitive closure, see “Enabling and Disabling Join Transitive Closure” on page 20-13.

Predicate Transformation and Factoring

Predicate factoring and transformation improves the number of choices available to the optimizer. It adds optimizable clauses to a query by extracting clauses from blocks of predicates linked with **or** into clauses linked by **and**. These additional optimizable clauses mean that there are more access paths available for query execution. The original **or** predicates are retained to ensure query correctness.

Predicate Transformation Example

All optimizable clauses in this query are enclosed in the **or** clauses:

```
select p.pub_id, price
from publishers p, titles t
where (
    t.pub_id = p.pub_id
    and type = "travel"
    and price between 15 and 30
    and p.pub_id in ("P220", "P583", "P780")
    )
or  (
    t.pub_id = p.pub_id
    and type = "business"
    and price between 15 and 50
    and p.pub_id in ("P651", "P066", "P629")
    )
```

Predicate transformation pulls clauses linked with **and** from blocks of clauses linked with **or**, such as those shown above. It extracts only clauses that occur in all parenthesized blocks. If the example above had a clause in one of the blocks linked with **or** that did not appear in the other clause, that clause would not be extracted.

Predicate Transformation Steps

During predicate transformation:

1. Simple predicates (joins, search arguments, and **in** lists) that are an exact match in each **or** clause are extracted. In the sample query, this clause matches exactly in each block, so it is extracted:

   ```
   t.pub_id = p.pub_id
   ```

   between clauses are converted to greater-than-or-equal and less-than-or-equal clauses before predicate transformation. The sample query above uses **between** 15 in both query blocks (though the end ranges are different). The equivalent clause is extracted by step 1:

   ```
   price >=15
   ```

2. Search arguments on the same table are extracted; all terms that reference the same table are treated as a single predicate during

expansion. Both *type* and *price* are columns in the *titles* table, so the extracted clauses are:

```
(type = "travel" and price >=15 and price <= 30)
or
(type = "business" and price >= 15 and price <= 50)
```

3. **in** lists and **or** clauses are extracted. If there are multiple **in** lists for a table within one of the blocks, only the first is extracted. The extracted lists for the sample query are:

```
 p.pub_id in ("P220", "P583", "P780")
or
 p.pub_id in ("P651", "P066", "P629")
```

4. These steps can overlap and extract the same clause, so any duplicates are eliminated.
5. Each generated term is examined to determine whether it can be used as an optimizable search argument or a join clause. Only those terms that are useful in query optimization are retained.
6. The additional clauses are added to the existing query clauses that were specified by the user.

Search Arguments and Useful Indexes

It is important to distinguish between **where** and **having** clause predicates that can be used to optimize the query, and those that are used later during query processing to filter the rows to be returned.

Search arguments can be used to determine the access path to the data rows when a column in the **where** clause matches a leading index key. The index can be used to locate and retrieve the matching data rows. Once the row has been located in the data cache or has been read into the data cache from disk, any remaining clauses are applied.

For example, if the *authors* table has on an index on *au_lname* and another on *city*, either index can be used to locate the matching rows for this query:

```
select au_lname, city, state
from authors
where city = "Washington"
and au_lname = "Catmull"
```

The optimizer uses statistics, including histograms, the number of rows in the table, the index heights, and the cluster ratios for the index and data pages to determine which index provides the cheapest access. The index that provides the cheapest access to the data pages is chosen and used to execute the query, and the other clause is applied to the data rows once they have been accessed.

Search Argument Syntax

Search arguments (SARGs) are expressions in one of these forms:

<*column*> <*operator*> <*expression*>

<*expression*> <*operator*> <*column*>

<*column*> is null

Where:

- *column* is only a column name. If functions, expressions, or concatenation are added to the column name, an index on the column cannot be used.
- *operator* must be one of the following:

  =, >, <, >=, <=, !>, !<, <>, !=, is null
- *expression* is either 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 known at compile time:
  - If *expression* is a known constant or can be converted to a known constant during preprocessing, it can be compared to the histogram values stored for an index to return accurate row estimates.
  - If the value of *expression* is not known at compile time, the optimizer uses the total density to estimate the number of rows to be returned by the query. The value of variables set in a query batch or parameters set within a stored procedure cannot be known until execution time.
  - If the datatype of the expression is not compatible with the datatype of the column, an index cannot be used, and is not considered. See "Datatype Mismatches and Query Optimization" on page 5-21 for more information.

Nonequality Operators

The nonequality operators, **<>** and **!=**, are special cases. The optimizer checks for covering nonclustered indexes if the column is indexed and uses a nonmatching index scan if an index covers the query.

However, if the index does not cover the query, the table is accessed via a table scan.

Examples of SARGs

The following are some examples of fully optimizable clauses. If there are statistics on these columns, they can be used to help estimate the number of rows the query will return. If there are indexes on the columns, the indexes can be used to access the data:

```
au_lname = "Bennett"

price >= $12.00

advance > $10000 and advance < $20000

au_lname like "Ben%" and price > $12.00
```

The following search arguments are **not** optimizable:

```
advance * 2 = 5000  /*expression on column side
                     not permitted */

substring(au_lname,1,3) = "Ben" /* function on
                                column name */
```

Note that these two clauses become optimizable if written in the form:

```
advance = 5000/2

au_lname like "Ben%"
```

Consider this query, with the only index on *au_lname*:

```
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 the *city* column. In this case, the index on *au_lname* would be used for the query. All data pages with a matching last name are brought into cache, and each matching row is examined to see if the city matches the search criteria.

How Statistics Are Used for SARGS

When you create an index, statistics are generated and stored in system tables. Some of the statistics relevant to determining the cost of search arguments and joins are:

- Statistics about the index: the number of pages and rows, the height of the index, the number of leaf pages, the average leaf row size.
- Statistics about the data in the column:
  - A histogram for the leading column of the index. Histograms are used to determine the selectivity of the SARG, that is, how many rows from the table match a given value.
  - Density values, measuring the density of keys in the index.
- Cluster ratios that measure the fragmentation of data storage and the effectiveness of large I/O.

Only a subset of these statistics (the number of leaf pages, for example) are maintained during query processing. Other statistics are updated only when you run **update statistics** or when you drop and re-create the index. These statistics can be displayed using **optdiag**. See Chapter 19, “Statistics Tables and Displaying Statistics with optdiag.”

Histogram Cells

When you create an index, a histogram is created on the first column of the index. The histogram stores information about the distribution of values in the column. You can also use **update statistics** to generate statistics for the minor keys of a compound index and columns used in unindexed search clauses.

The histogram for a column contains data in a set of steps or cells. The number of cells can be specified when the index is created or when the **update statistics** command is run. For each cell, the histogram stores a column value and a weight for the cell.

There are two types of cells in histograms:

- A **frequency cell** represents a value that has a high proportion of duplicates in the column. The weight of a frequency cell times the number of rows in the table equals the number of rows in the table that match the value for the cell. If a column does not have highly duplicated values, there are only range cells in the histogram.
- **Range cells** represent a range of values. Range cell weights and the range cell density are used for estimating the number of rows to be returned when search argument values falls within a range cell.

For more information on histograms, see "Histogram Displays" on page 19-15.

Density Values

Density is a measure of the average proportion of duplicate keys in the index. It varies between 0 and 1. An index with *N* rows whose keys are unique has a density of 1 / *N*; an index whose keys are all duplicates of each other has a density of 1.

For indexes with multiple keys, density values are computed and stored for each prefix of keys in the index. That is, for an index on columns A, B, C, D, densities are stored for:

- A
- A, B
- A, B, C
- A, B, C, D

Range Cell Density and Total Density

For each prefix subset, two density values are stored:

- Range cell density, used for search arguments
- Total density, used for joins

Range cell density represents the average number of duplicates of all values that are represented by range cells in the histogram. Total density represents the average number of duplicates for all values, those in both frequency and range cells. Total density is used to estimate the number of matching rows for joins and for search arguments whose value is not known when the query is optimized.

How the Optimizer Uses Densities and Histograms

When the optimizer analyzes a SARG, it uses the histogram values, densities, and the number of rows in the table to estimate the number of rows that match the value specified in the SARG:

- If the SARG value matches a frequency cell, the estimated number of matching rows is equal to the weight of the frequency cell multiplied by the number of rows in the table. This query includes a data value with a high number of duplicates, so it matches a frequency cell:

  ```
  where authors.city = "New York"
  ```

  If the weight of the frequency cell is .015606, and the *authors* table has 5000 rows, the optimizer estimates that the query returns 5000 * .015606 = 78 rows.

- If the SARG value falls within a range cell, the optimizer uses the range cell density to estimate the number of rows. For example, a query on a *city* value that falls in a range cell, with a range cell density of .000586 for the column, would estimate that 5000 * .000586 = 3 rows would be returned.
- For range queries, the optimizer adds the weights of all cells spanned by the range of values. When the beginning or end of the range falls in a range cell, the optimizer uses interpolation to estimate the number of rows from that cell that are included in the range.

Using Statistics on Multiple Search Arguments

When there are multiple search arguments on the same table, the optimizer uses statistics to combine the selectivity of the search arguments.

This query specifies search arguments for two columns in the table:

```
select title_id
from titles
where type = "news"
and price < $20
```

With an index on *type, price*, the selectivity estimates vary, depending on whether statistics have been created for *price*:

- With only statistics for *type*, the optimizer uses the frequency cell weight for *type* and a default selectivity for *price*. The selectivity for *type* is .106600, and the default selectivity for an open-ended range query is 33%. The number of rows to be returned for the query is estimated using .106600 * .33, or .035178. With 5000 rows in the table, the estimate is 171 rows. See Table 5-2 for the default values used when statistics are not available.

- With statistics added for *price*, the histogram is used to estimate that .133334 the rows match the search argument on *price*. Multiplied by the selectivity of *type*, the result is .014213, and the row estimate is 71 rows.

The actual number of rows returned is 53 rows for this query, so the additional statistics improved the accuracy. For this simple single-table query, the more accurate selectivity did not change the access method, the index on *type, price*. For some single-table queries, however, the additional statistics can help the optimizer make a better choice between using a table scan or using other indexes. In join queries, having more accurate statistics on each table can result in more efficient join orders.

Default Values for Search Arguments

When statistics are not available for a search argument or when the value of a search argument is not known at optimization, the optimizer uses default values. These values are shown in Table 5-2.

Table 5-2: Density approximations for unknown search arguments

Operation Type	Operator	Density Approximation
Equality	**=**	Total density, if statistics are available for the column, or 10%
Open-ended range	**<**, **<=**, **>**, or **>=**	33%
Closed range	**between**	25%

SARGs Using Variables and Parameters

Since the optimizer computes its estimates before a query executes, it cannot know the value of a variable that is set in the batch or procedure. If the value of a variable is not known at compile time, the optimizer uses the default values shown inTable 5-2

For example, the value of *@city* is set in this batch:

```
declare @city varchar(25)
select @city = city from publishers
    where pub_name = "Brave Books"
select au_lname from authors where city = @city
```

The optimizer uses the total density, .000879, and estimates that 4 row will be returned; the actual number of rows could be far larger.

A similar problem exists when you set the values of variables inside a stored procedure. In this case, you can improve performance by splitting the procedure: set the variable in the first procedure and then call the second procedure, passing the variables as parameters. The second procedure can then be optimized correctly. See “Splitting Stored Procedures to Improve Costing” on page 5-30 for an example.

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. When possible, move functions and other operations to the expression side of the clause.
- Avoid incompatible datatypes for columns that will be joined and for variables and parameter used as search arguments. See “Datatype Mismatches and Query Optimization” on page 5-21 for more information.
- Use the leading column of a composite index as a search argument. 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 mare than 102 predicates for a table, put the most potentially useful clauses near the beginning of the query, since only the first 102 SARGs on each table are used during optimization. (All of the search conditions are used to qualify the rows.)
- Some queries using > (greater than) may perform better if you can rewrite them to use >= (greater than or equal to). For example, 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 more efficient to write the query like this:

```
select * from table where int_col >= 4
```

This optimization is more difficult with character strings and floating-point data. You need to know your data.

- Check **showplan** output to see which keys and indexes are used.
- If you expect an index is not being used when you expect it to be, check **dbcc traceon(302)** output to see if the optimizer is considering the index.

Join Syntax and Join Processing

Join clauses take the form:

```
table1.column_name <operator> table2.column_name
```

The join operators are:

```
=, >, >=, <, <=, !>, !<, !=, <>, *=, =*
```

And:

```
table1 [ left | right ] join table2
    on column_name = column_name

table1 inner join table2
    on column_name = column_name
```

When joins are optimized, the optimizer can only consider indexes on column names. Any type of operator or expression in combination with the column name means that the optimizer does not evaluate using an index on the column as a possible access method. If the columns in the join are of incompatible datatypes, the optimizer can only consider an index on one of the columns.

How Joins Are Processed

When the optimizer creates a query plan for a join query:

- It evaluates indexes for each table by estimating the I/O required for each possible index and for a table scan.
- It determines the join order, basing the decision on the total cost estimates for the possible join orders. It estimates costs for both nested-loop joins and sort-merge joins.

- 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 "The Reformatting Strategy" on page 7-21.
- It determines the I/O size and caching strategy.
- It also compares the cost of serial and parallel execution, if parallel query processing is enabled. See Chapter 12, "Parallel Query Optimization," for more information.

Factors that determine costs on single-table selects, such as appropriate indexing, search argument selectivity, and density of keys, become much more critical for joins.

When Statistics Are Not Available for Joins

If statistics are not available for a column in a join, the optimizer uses default values:

Operator Type	Examples	Default selectivity
Equality	`t1.c1 = t1.c2`	1/rows in smaller table

Nonequality	`t1.c1 > t1.c2`
	`t1.c1 >= t1.c2`
	`t1.c1 < t1.c2`
	`t1.c1 <= t1.c2`
		33%

For example, in the following query, the optimizer uses 1/500 for the join selectivity for both tables if there are no statistics for either *city* column, and *stores* has 500 rows and *authors* has 5000 rows:

```
select au_fname, au_lname, stor_name
    from authors a, stores s
    where a.city = s.city
```

Density Values and Joins

When statistics are available on a join column, the total density is used to estimate how many rows match each join key. If the *authors* table has 5000 rows, and the total density for the *city* column is .000879, the optimizer estimates that 5000 * .000879 = 4 rows will be returned from *authors* each time a join on the *city* column matches a row from the other table.

Multicolumn Joins

When a join query specifies multiple join columns on two tables, and there is a composite index on the columns, the composite total density is used. For example, if *authors* and *publishers* each has an index on *city, state*, the composite total density for *city, state* is used for each table in this query:

```
select au_lname, pub_name
from authors a, publishers p
where a.city = p.city
and a.state = p.state
```

Search Arguments and Joins on a Table

When there are search arguments and joins on a table, the selectivities of the columns are combined during join costing to estimate the number of rows more accurately.

The following example joins *authors* and *stores* on both the *city* and *state* columns. There is a search argument on *authors.state*, so search argument transitive closure adds the search argument for *stores.state* table also:

```
select au_fname, au_lname, stor_name
from authors a, stores s
where  a.city = s.city
and a.state = s.state
and a.state = "GA"
```

If there is an index on *city* for each table, but no statistics available for *state*, the optimizer uses the default search argument selectivity (10%) combined with the total density for *city*. This overestimates the number of rows that match the search argument for this query, for a state with more rows that match a search argument on state, it would underestimate the number of rows. When statistics exist for *state* on each table, the estimate of the number of qualifying rows improves, and overall costing for the join query improves also.

Datatype Mismatches and Query Optimization

One common problem when queries fail to use indexes as expected is datatype mismatches. Datatype mismatches occur:

- With search clauses using variables or stored procedure parameters that have a different datatype than the column, for example:

```
where int_col = @money_parameter
```

- In join queries when the columns being joined have different datatypes, for example:

```
where tableA.int_col = tableB.money_col
```

Datatype mismatches lead to optimization problems when they prevent the optimizer from considering an index. The most common problems arise from:

- Comparisons between the integer types, *int*, *smallint* and *tinyint*
- Comparisons between *money* and *smallmoney*
- Comparisons between *datetime* and *smalldatetime*
- Comparisons between *numeric* and *decimal* types of differing precision and scale
- Comparisons between *numeric* or *decimal* types and integer or money columns

To avoid problems, use the same datatype (including the same precision and scale) for columns that are likely join candidates when you create tables. Use a matching datatype for any variables or stored procedure parameters used as search arguments. The following sections detail the rules and considerations applied when the same datatype is not used, and provide some troubleshooting tips.

Overview of the Datatype Hierarchy and Index Issues

The datatype hierarchy controls the use of indexes when search arguments or join columns have different datatypes. The following query prints the *hierarchy* values and datatype names:

```
select hierarchy, name from systypes order by 1
```

```
hierarchy name
 --------- ------------------------------
         1 floatn
         2 float
         3 datetimn
         4 datetime
         5 real
         6 numericn
         7 numeric
         8 decimaln
         9 decimal
        10 moneyn
```

```
11 money
12 smallmoney
13 smalldatetime
14 intn
15 int
16 smallint
17 tinyint
18 bit
19 sysname
19 varchar
19 nvarchar
20 char
20 nchar
21 timestamp
21 varbinary
22 binary
23 text
24 image
99 extended type
```

If you have created user-defined datatypes, they are also listed in the query output, with the corresponding hierarchy values.

General Rules for Index Use

The general rule is that when different datatypes are used, the *systypes.hierarchy* value determines whether an index can be used.

- For search arguments, the index is considered when the column's datatype is same as, or precedes, the *hierarchy* value of the parameter or variable.
- For a join, the index is considered only on the column whose *systypes.hierarchy* value is the same as the other column's, or precedes the other column's in the hierarchy.

Exceptions to the Rule

The exceptions are:

- Comparisons between *char* and *varchar* datatypes or between *binary* and *varbinary* datatypes. For example, although their hierarchy values are 19 and 20 respectively, *char* and *varchar* columns are treated as the same datatype for index consideration purposes. The index is considered for both columns in this join:

  ```
  where t1.char_column = t2.varchar_column
  ```

char columns that accept NULL values are stored as *varchar*, but indexes can still be used on both columns for joins.

- The null type of the column has no effect, that is, although *float* and *floatn* have different hierarchy values, they are always treated as the same datatype.
- Comparisons of *decimal* or *numeric* types also take precision and scale into account. This includes comparisons of *numeric* or *decimal* types to each other, and comparisons of *numeric* or *decimal* to other datatypes such as *int* or *money*. See "Comparison of Numeric and Decimal Datatypes" on page 5-24 for more information.

Comparison of Numeric and Decimal Datatypes

When a query joins columns of *numeric* or *decimal* datatypes, an index can be used when both of these conditions are true:

- The scale of the column being considered for a join equals or exceeds the scale of the other join column, and
- The length of the integer portion of the column equals or exceeds the length of the other column's integer portion.

Here are some examples of when indexes can be considered:

Datatypes in the Join	Indexes Considered
numeric(12,4) and *numeric(16,4)*	Index considered only for *numeric(16,4)*, the integer portion of *numeric(12,4)* is smaller.
numeric(12,4) and *numeric(12,8)*	Neither index is considered, integer portion is smaller for *numeric(12,8)* and scale is smaller for *numeric(12,4)*.
numeric(12,4) and *numeric(12,4)*	Both indexes are considered.

Comparing Numeric Types to Other Datatypes

When comparing *numeric* and *decimal* columns to columns of other numeric datatypes, such as *money* or *int*:

- *numeric* and *decimal* precede integer and money columns in the hierarchy, so the index on the *numeric* or *decimal* column is the only index considered.

- The precision and scale requirements must be met in order for the *numeric* or *decimal* index to be considered. The scale of the *numeric* column must be equal to, or greater than, the scale of the integer or money column, and the number of digits in the integer portion of the numeric column must be equal to or greater than the maximum number of digits usable for the integer or money column.

The precision and scale of integer and money types is shown in Table 5-3.

Table 5-3: Precision and scale of integer and money types

Datatype	Precision, Scale
tinyint	3,0
smallint	5,0
int	10,0
smallmoney	10,4
money	19,4

Datatypes for Parameters and Variables Used as SARGs

When declaring datatypes for variables or stored procedure parameters to be used as search arguments, match the datatype of the column in the variable or parameter declaration to ensure the use of an index. For example:

```
declare @int_var int
select @int_var = 50
select *
from t1
where int_col = @int_var
```

Use of the index depends on the precedence of datatypes in the hierarchy. The index on a column can be used only if the column's datatype precedes the variable's datatype. For example, *int* precedes *smallint* and *tinyint* in the hierarchy. Here are just the integer types:

```
hierarchy name
 --------- ------------------------------
        15 int
        16 smallint
        17 tinyint
```

If a variable or parameter has a datatype of *smallint* or *tinyint*, an index on an *int* column can be used for a query. But an index on a *tinyint* column cannot be used for an *int* parameter.

Similarly, *money* precedes *int*. If a variable or parameter of *money* is compared to an *int* column, an index on the *int* column cannot be used.

This eliminates issues that could arise from truncation or overflow. For example, it would not be useful or correct to attempt to truncate the *money* value to 5 in order to use an index on *int_col* for this query:

```
declare @money_var money
select @money_var = $5.12
select * from t1 where int_col = @money_var
```

Troubleshooting Datatype Mismatch Problems for SARGs

If there is a datatype mismatch problem with a search argument on an indexed column, the query can use another index if there are other search arguments or it can perform a table scan. **showplan** output displays the access method and keys used for each table in a query.

You can use **dbcc traceon(302)** to determine whether an index is being considered. For example, using an integer variable as a search argument on *int_col* produces the following output:

```
Selecting best index for the SEARCH CLAUSE:
    t1.int_col = unknown-value

SARG is a local variable or the result of a
function or an expression, using the total density
to estimate selectivity.

Estimated selectivity for int_col,
   selectivity = 0.020000.
```

Using an incompatible datatype such a *money* for a variable used as a search argument on an integer column does not produce a “Selecting best index for the SEARCH CLAUSE” block in **dbcc traceon(302)** output, indicating that the index is not being considered, and cannot be used. If an index is not used as you expect in a query, looking for this costing section in **dbcc traceon(302)** output should be one of your first debugging steps.

The “unknown-value” and the fact that the total density is used to estimate the number of rows that match this search argument is due to the fact that the value of the variable was set in the batch; it is not a datatype mismatch problem.See “SARGs Using Variables and Parameters” on page 5-17 for more information.

Compatible Datatypes for Join Columns

The optimizer only considers an index for joined columns when the column types are the same or when the datatype of the join column precedes the other column’s datatype in the datatype hierarchy. This means that the optimizer considers using the index on only one of the join columns, limiting the choice of join orders.

For example, this query joins columns of *decimal* and *int* datatypes:

```
select *
from t1, t2
where t1.decimal_col = t2.int_col
```

decimal precedes *int* in the hierarchy, so the optimizer can consider an index on *t1.decimal_col*, but cannot use an index on *t2.int_col*. The result is likely to be a table scan of *t2*, followed by use of the index on *t1.decimal_col*.

Table 5-4 shows how the hierarchy affects index choice for some commonly problematic datatypes.

Table 5-4: Indexes considered for mismatched column datatypes

Join Column Types	Index Considered on Column of Type
money and *smallmoney*	*money*
datetime and *smalldatetime*	*datetime*
int and *smallint*	*int*
int and *tinyint*	*int*
smallint and *tinyint*	*smallint*

Troubleshooting Datatype Mismatch Problems for Joins

If you suspect that an index is not being considered on one side of a join due to datatype mismatches, use **dbcc traceon(302)**. In the output, look for the “Selecting best index for the JOIN CLAUSE”. If datatypes are compatible, you see two of these blocks for each join; for example:

```
Selecting best index for the JOIN CLAUSE:
    t1.int_col = t2.int_col
```

And later in the output for the other table in the join:

```
Selecting best index for the JOIN CLAUSE:
    t2.int_col = t1.int_col
```

For a query that compares incompatible datatypes, for example, comparing a *decimal* column to an *int*, column, there is only the single block:

```
Selecting best index for the JOIN CLAUSE:
    t1.decimal_col = t2.int_col
```

This means that the join costing for using an index with *t2.int_col* as the outer column is not performed.

Suggestions on Datatypes and Comparisons

These suggestions can help avoid datatype mismatch problems:

- When you create tables, use the same datatypes for columns that will be joined.
- If columns of two frequently-joined tables have different datatypes, consider using **alter table...modify** to change the datatype of one of the columns.
- Use the column's datatype whenever declaring variables or stored procedure parameters that will be used as search arguments.
- Consider user-defined datatype definitions. Once the definitions are created with **sp_addtype**, they can be used in commands such **create table**, **alter table**, and **create procedure**, and can be used for datatype declarations.
- For some queries where datatype mismatches cause performance problems, you may be able to use the convert function so that indexes are considered on the other table in the join. The section below describes this workaround.

Forcing a Conversion to the Other Side of a 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 for some queries. In the following query, an index on *smallmoney_col* cannot be used, so the query performs a table scan on *huge_table*:

```
select *
from tiny_table, huge_table
where tiny_table.money_col =
    huge_table.smallmoney_col
```

Performance improves if the index on *huge_table.smallmoney_col* can be used. Using the **convert** function on the *money* column of the small table allows the index on the large table to be used, and a table scan is performed on the small table:

```
select *
from tiny_table, huge_table
where convert(smallmoney,tiny_table.money_col) =
    huge_table.smallmoney_col
```

This workaround assumes that there are no values in *tinytable.money_col* that are large enough to cause datatype conversion errors during the conversion to *smallmoney*. If there are values larger than the maximum value for *smallmoney*, you can salvage this solution by adding a search argument specifying the maximum values for a *smallmoney* column:

```
select smallmoney_col, money_col
from tiny_table , huge_table
where convert(smallmoney,tiny_table.money_col) =
    huge_table.smallmoney_col
and tiny_table.money_col <= 214748.3647
```

Be careful with floating-point and numeric data. Conversion can change the meaning of some queries. This query compares integers and floating-point numbers:

```
select *
    from tab1, tab2
    where tab1.int_column = tab2.float_column
```

In the query above, an index on *int_column* cannot be used. This conversion forces the index access to *tab1*, but also returns different results than the query that does not use **convert**:

```
select *
from tab1, tab2
where tab1.int_col = convert(int, tab2.float_col)
```

For example, if *int_column* is 4, and *float_column* is 4.2, the modified query implicitly converts it to 4, and returns a row not returned by the original query. The workaround can be salvaged by adding this self-join:

```
and tab2.float_col = convert(int, tab2.float_col)
```

This workaround assumes that all values in *tab2.float_col* can be converted to *int* without conversion errors.

Splitting Stored Procedures to Improve Costing

The optimizer cannot use statistics the final **select** in the following procedure, because it cannot know the value of *@city* until execution time:

```
create procedure au_city_names
    @pub_name varchar(30)
as
    declare @city varchar(25)
    select @city = city
    from publishers where pub_name = @pub_name
    select au_lname
        from authors
        where city = @city
```

The following example shows the procedure split into two procedures. The first procedure calls the second one:

```
create procedure au_names_proc
    @pub_name varchar(30)
as
    declare @city varchar(25)
    select @city = city
        from publishers
        where pub_name = @pub_name
    exec select_proc @city

create procedure select_proc @city varchar(25)
as
    select au_lname
        from authors
        where city = @city
```

When the second procedure executes, Adaptive Server knows the value of *@city* and can optimize the **select** statement. Of course, if you modify the value of *@city* 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 *@city* at the start of the procedure. If *@city* has different values each time the second procedure is executed, leading to very different query plans, you may want to use **with recompile**.

6 Access Methods and Query Costing for Single Tables

This chapter introduces the methods that Adaptive Server uses to access rows in tables. It examines various types of queries on single tables, and describes the access methods that can be used, and the associated costs.

This chapter contains the following sections:

Chapter 5, “Understanding the Query Optimizer,” explains how the optimizer uses search arguments and join clauses to estimate the number of rows that a query will return. This chapter looks at how the optimizer uses row estimates and other statistics to estimate the number of pages that must be read for the query, and how many logical and physical I/Os are required.

This chapter looks at queries that affect a single table. For queries that involve more than one table, see Chapter 7, “Access Methods and Query Costing for Joins and Subqueries.” For parallel queries, see Chapter 12, “Parallel Query Optimization.”

How To Use This Chapter While Analyzing Queries

This chapter contains information about query processing that you can use at in several ways:

- This chapter provides a general overview of the access methods that Adaptive Server uses to process a variety of queries, including illustrations and sample queries. This information will help you understand how particular types of queries are executed and how query performance can be improved by adding indexes or statistics for columns used in the queries.

- This chapter also provides a description of how the optimizer arrives at the logical and physical I/O estimates for the queries. These descriptions can help you understand whether the I/O use and response time are reasonable for a given query. These descriptions can be used with the following tuning tools:
  - **optdiag** can be used to display the statistics about your tables, indexes, and column values. See Chapter 19, "Statistics Tables and Displaying Statistics with optdiag."
  - **showplan** displays the access method (table scan, index scan, type of OR strategy, and so forth) for a query. See Chapter 17, "Using set showplan."
  - **statistics io** displays the logical and physical I/O for each table in a query. See Chapter 6, "Access Methods and Query Costing for Single Tables."
- This chapter provides detailed formulas, very close to the actual formulas used by Adaptive Server. These formulas are meant to be used in conjunction with the tuning tools described in later chapters:
  - **optdiag** can be used to display the statistics that you need to apply the formulas. Chapter 19, "Statistics Tables and Displaying Statistics with optdiag."
  - **dbcc traceon(302)** displays the sizes, densities, selectivities and cluster ratios used to produce logical I/O estimates, and **dbcc traceon(310)** displays the final query costing for each table, including the estimated physical I/O. Chapter 18, "Tuning with dbcc traceon."

  In many cases, you will only need to use these formulas when you are debugging problem queries. You may need to discover why an **or** query performs a table scan, or why an index that you thought was useful is not being used by a query.

This chapter can also help you determine when to stop working to improve the performance of a particular query. If you know that it needs to read a certain number of index pages and data pages, and the number of I/Os cannot be reduced further by adding a covering index, you know that you have reached the optimum performance possible for query analysis and index selection. You might need to look at other issues, such as cache configuration, parallel query options, or object placement.

Basic Units of Costing

When the optimizer estimates costs for the query, the two factors it considers are the cost of physical I/O, reading pages from disk, and the cost of logical I/O, finding pages in the data cache. The optimizer assigns 18 as the cost of a physical I/O and 2 as the cost of a logical I/O. These are relative units of cost and do not represent time units such as milliseconds or clock ticks. These units are used in the formulas in this chapter, with the physical I/O costs first, then the logical I/O costs. The total cost of accessing a table can be expressed as:

```
Cost = All physical IOs * 18 + All logical IOs * 2
```

Table Scan Cost

When a query requires a table scan, Adaptive Server reads each page of the table from disk into the data cache and checks the data values (if there is a **where** clause) and returns qualifying rows.

Table scans are performed:

- When no index exists on the columns used in the search clauses.
- When the optimizer determines that using the index is more expensive than performing a table scan. The optimizer may determine that it is cheaper to read the data pages directly than to read the index pages and then the data pages for each row that is to be returned.

The cost of a table scan depends on the size of the table and the I/O size.

Cost of a Table Scan on an Allpages-Locked Table

The I/O cost of a table scan on an allpages-locked table using 2K I/O is one physical I/O and one logical I/O for each page in the table:

```
Table scan cost =    Number of pages * 18
                   + Number of pages * 2
```

If the table uses a cache with large I/O, the number of physical I/Os is estimated by dividing the number of pages by the I/O size and using a factor that is based on the data page cluster ratio to estimate the number of large I/Os that need to be performed. Since large I/O cannot be performed on any data pages on the first extent in the allocation unit, each of those pages must be read with 2K I/O.

The logical I/O cost is one logical I/O for each page in the table. The formula is:

```
Table scan cost =    (Pages /Pages per IO) * Clustering adjustment* 18
                   + Number of pages * 2
```

See "How Cluster Ratios Affect Large I/O Estimates" on page 6-7 for more information on cluster ratios.

➤ ***Note***

Adaptive Server does not track the number of pages in the first extent of an allocation unit for an allpages-locked table, so the optimizer does not include this slight additional I/O in its estimates.

Cost of a Table Scan on a Data-Only-Locked Tables

Tables that use data-only locking do not have page chains like allpages-locked tables. To perform a table scan on a data-only-locked table, Adaptive Server:

- Reads the OAM (object allocation map) page(s) for the table
- Uses the pointers on the OAM page to access the allocation pages
- Uses the pointers on the allocation pages to locate the extents used by the table
- Performs either large I/O or 2K I/O on the pages in the extent

The total cost of a table scan on a data-only-locked table includes the logical and physical I/O for all pages in the table, plus the cost of logical and physical I/O for the OAM and allocation pages.

Figure 6-1 shows the pointers from OAM pages to allocation pages and from allocation pages to extents.

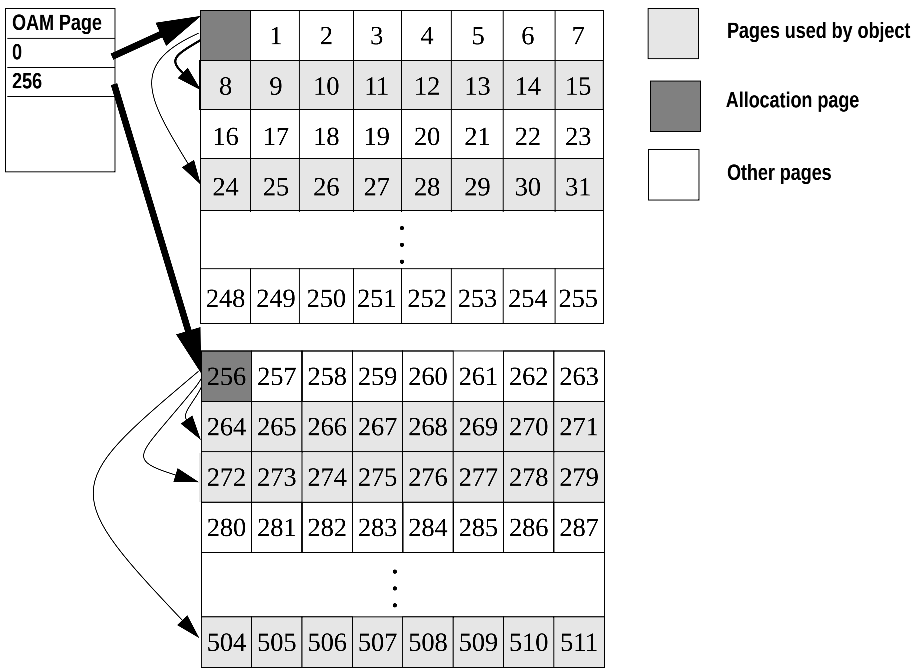


Figure 6-1: Sequence of pointers for OAM scans

The formula for computing the cost of an OAM scan with 2K I/O is:

```
OAM Scan Cost =   (OAM_alloc_pages + Num_pages) * 18
                + (OAM_alloc_pages + Num_pages)* 2
```

When large I/O can be used, the optimizer adds the cost of performing 2K I/O for the pages in the first extent of each allocation unit to the cost of performing 16K I/O on the pages in regular extents. The number of physical I/Os is the number of pages in the table, modified by a cluster adjustment that is based on the data page cluster ratio for the table. See "How Cluster Ratios Affect Large I/O Estimates" on page 6-7 for more information on cluster ratios.

Logical I/O costs are one I/O per page in the table, plus the logical I/O cost of reading the OAM and allocation pages. The formula for computing the cost of an OAM scan with large I/O is:

```
OAM Scan Cost =   OAM_alloc_pages * 18
                + Pages in 1st extent * 18
                + Pages in other extents / Pages per IO * Cluster adjustment * 18
                + OAM_alloc_pages * 2
                + Pages in table * 2
```

optdiag reports the number of pages for each of the needed values.

Added Cost for Forwarded Rows

When a data-only-locked table contains forwarded rows, the I/O cost of reading the forwarded rows is added to the logical and physical I/O for a table scan. See "Updates on Allpages-Locked Heap Tables" on page 3-18 for more information on row forwarding.

From Rows to Pages

When the optimizer costs the use of an index to resolve a query, it first estimates the number of qualifying rows, and then estimates the number of pages that need to be read.

The examples in Chapter 5, "Understanding the Query Optimizer," show how Adaptive Server estimates the number of rows for a search argument or join using statistics. Once the number of rows has been estimated, the optimizer estimates the number of data pages and index leaf pages that need to be read:

- For tables, the optimizer divides the number of rows in the table by the number of pages to determine the average number of rows per data page.
- To estimate the average number of rows per page on the leaf level of an index, the optimizer divides the number of rows in the table by the number of leaf pages in the index.

The formulas are:

```
Average rows per data page = Rows in table / Pages in table
Average rows per leaf page = Rows in table / Pages in leaf level
```

After the number of pages is estimated, data page and index page cluster ratios are used to adjust the page estimates for queries using large I/O, and data row cluster ratios are used to estimate the number of data pages for queries using noncovering indexes.

How Cluster Ratios Affect Large I/O Estimates

When clustering is high, large I/O is effective. As the cluster ratios decline, effectiveness of large I/O drops rapidly. To refine I/O estimates, the optimizer uses a set of cluster ratios:

- For a table, the data page cluster ratio measures the packing and sequencing of pages on extents.
- For an index, the data page cluster ratio measures the effectiveness of large I/O for accessing the table using this index.
- The index page cluster ratio measures the packing and sequencing of leaf-level index pages on index extents.

➤ ***Note***

The data row cluster ratio, another cluster ratio used by query optimization, is used to cost the number of data pages that need to be accessed during scans using a particular index. It is not used in large I/O costing. See "How Data Row Cluster Ratio Refines Page Estimates" on page 6-13 for more information.

optdiag displays the cluster ratios for tables and indexes.

The Data Page Cluster Ratio

The data page cluster ratio for a table measures the effectiveness of large I/O for table scans. Its use is slightly different depending on the locking scheme.

Data Page Cluster Ratios on Allpages-Locked Tables

For allpages-locked tables, a table scan or a scan that uses a clustered index to scan many pages follows the next-page pointers on each data page. Immediately after the clustered index is created, the data page cluster ratio is 1.0, and pages are ordered by page number on the extents. However, after updates and page splits, the page chain can be fragmented across the page chain, as shown in Figure 6-2, where page 10 has been split; the page pointers point from page 10 to page 26 in another extent, then to page 11.

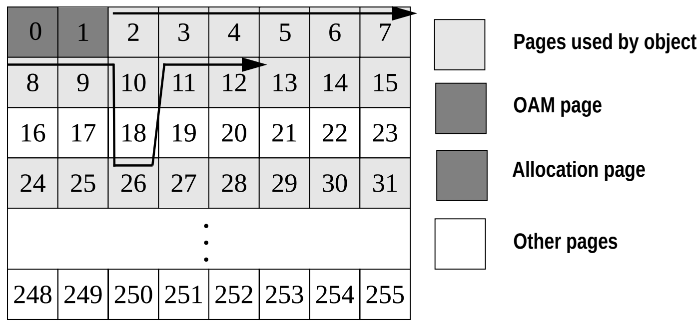


Figure 6-2: Page chain crossing extents in an allpages-locked table

The data page cluster ratio for an allpages-locked table measures the effectiveness of large I/O for both table scans and clustered index scans.

Data Page Cluster Ratio on Data-Only-Locked Tables

For data-only-locked tables, the data page cluster ratio measures how well the pages are packed on the extents. A cluster ratio of 1.0 indicates complete packing of extents, with the page chain ordered. If extents contain unused pages, the data page cluster ratio is less than 1.0.

optdiag reports two data page cluster ratios for data-only-locked tables with clustered indexes. The value reported for the table is used for table scans. The value reported for the clustered index is used for scans using the index.

Index Page Cluster Ratio

The index page cluster ratio measures the packing and sequencing of index leaf pages on extents for nonclustered indexes and clustered indexes on data-only-locked tables. For queries that need to read more than one leaf page, the leaf level of the index is scanned using next-page or previous page pointers. If many leaf rows need to be read, 16K I/O can be used on the leaf pages to read an extent at a time. The index page cluster ratio measures fragmentation of the page chain for the leaf level of the index.

Evaluating the Cost of Index Access

When a query has search arguments on useful indexes, the query accesses only the index pages and data pages that contain rows that match the search arguments. Adaptive Server compares the total cost of index and data page I/O to the cost of performing a table scan, and uses the cheapest method.

Evaluating the Cost of a Query That Returns a Single Row

A query that returns a single row using an index performs one I/O for each index level plus one read for the data page. The optimizer estimates the total cost as one physical I/O and one logical I/O for each index page and the data page. The cost for a point query is:

```
Point query cost  =  (Number of index levels + data page ) * 18
                  +  (Number of index levels + data page ) * 2
```

optdiag output displays the number of index levels.

The root page and intermediate pages of frequently used indexes are often found in cache. In that case, actual physical I/O is reduced by one or two reads.

Evaluating the Cost of a Query That Returns Many Rows

A query that returns many rows may be optimized very differently, depending on the type of index and the number of rows to be returned. Some examples are:

- Queries with search arguments that match many values, such as:

```
select title, price
from titles
where pub_id = "P099"
```

- Range queries, such as:

```
select title, price
from titles
where price between $20 and $50
```

For queries that return a large number of rows using the leading key of the index, clustered indexes and covering nonclustered indexes are very efficient:

- If the table uses allpages locking, and has a clustered index on the search arguments, the index is used to position the scan on the first qualifying row. The remaining qualifying rows are read by scanning forward on the data pages.

- If a nonclustered index or the clustered index on a data-only-locked table covers the query, the index is used to position the scan at the first qualifying row on the index leaf page, and the remaining qualifying rows are read by scanning forward on the leaf pages of the index.

If the index does not cover the query, using a clustered index on a data-only-locked table or a nonclustered index requires accessing the data page for each index row that matches the search arguments on the index. The matching rows may be scattered across many data pages, or they could be located on a very small number of pages, particularly if the index is a clustered index on a data-only-locked table. The optimizer uses data row cluster ratios to estimate how many physical and logical I/Os are required to read all of the qualifying data pages.

Range Queries Using Clustered Indexes (Allpages Locking)

To estimate the number of physical I/Os required for a range query using a clustered index on an allpages-locked table, the optimizer adds the physical and logical I/O for each index level and the physical and logical I/O of reading the needed data pages. Since data pages are read in order following the page chain, the cluster adjustment helps estimate the effectiveness of large I/O. The formula is:

```
Data pages =        Number of qualified rows / Data rows per page

Range query cost =  Number of index levels * 18
                 +  Data pages/pages per IO * Cluster adjustment * 18
                 +  Number of index levels * 2
```

If a query returns 500 rows, and the table has 10 rows per page, the query needs to read 50 data pages, plus one index page for each index level. If the query uses 2K I/O, it requires 50 I/Os for the data pages. If the query uses 16K I/O, these 50 data pages require 7 I/Os.

The cluster adjustment uses the data page cluster ratio to refine the estimate of large I/O for the table, based on how fragmented the data page storage has become on the table's extents.

Figure 6-3 shows how a range query using a clustered index positions the search on the first matching row on the data pages. The next-page pointers are used to scan forward on the data pages until a nonmatching row is encountered.

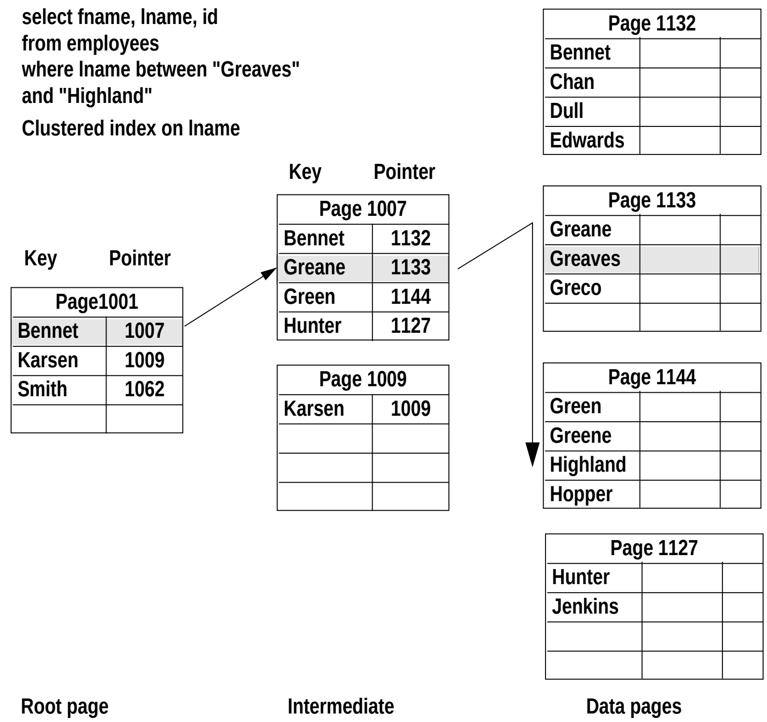


Figure 6-3: Range query on the clustered index of an allpages-locked table

Range Queries with Covering Indexes

Range queries using covering indexes perform very well because:

- The index is used to position the search at the first qualifying row on the index leaf level.
- 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 index is configured for large I/O, up to 8 leaf-level pages can be read per I/O.
- The data pages do not have to be accessed.

Both nonclustered indexes and clustered indexes on data-only-locked tables have a leaf level above the data level, so they can provide index covering.

Figure 6-4 shows a range query on a covering index on (*lname, fname, au_id*).

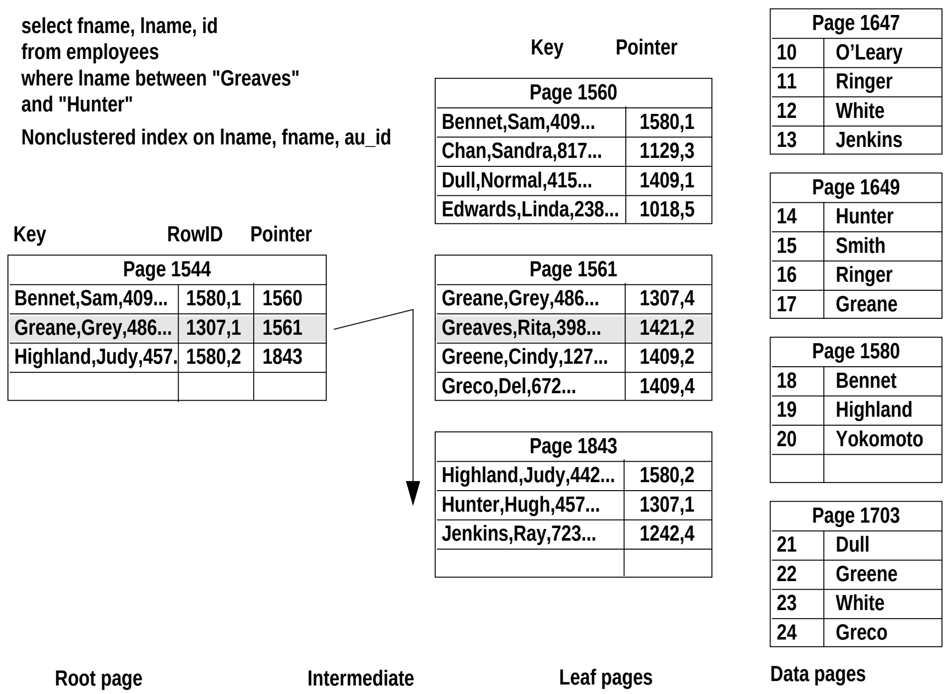


Figure 6-4: Range query with a covering index

The cost of using a covering index is determined by:

- The number of non-leaf index levels
- The number of rows that the query returns
- The number of rows per page on the leaf level of the index
- The number of leaf pages read per I/O
- The index page cluster ratio, used to adjust large I/O estimates when the index pages are not stored consecutively on the extents

This formula shows the costs:

```
Leaf pages = Number of qualified rows / Leaf level rows per page

Covered scan cost = Number of index levels * 18
                  + (Leaf pages /Pages per IO) * Cluster adjustment * 18
                  + Number of index levels * 2
                  + Leaf pages * 2
```

For example, if a query needs to read 1,200 leaf pages, and there are 40 rows per leaf-level page, the query needs to read 30 leaf-level pages. If large I/O can be used, this requires 4 I/Os. If inserts have caused page splits on the index leaf-level, the cluster adjustment increases the estimated number of large I/Os.

Range Queries with Noncovering Indexes

When a nonclustered index or a clustered index on a data-only-locked table does not cover the query, Adaptive Server:

- Uses the index to locate the first qualifying row at the leaf level of the nonclustered index
- Follows the pointer to the data page for that index, and reads the page
- Finds the next row on the index page, and locates its data page, and continues this process until all matching keys have been used

How Data Row Cluster Ratio Refines Page Estimates

For each subsequent key, the data row could be on the same page as the row for the previous key, or the data row may be on a different page in the table. The clustering of key values for each index is measured by a value called the **data row cluster ratio**. The data row cluster ratio is applied to estimate the number of logical and physical I/Os.

When the Data Row Cluster Ratio Is High

When the data row cluster ratio is 1.0, clustering is very high. High cluster ratios are always seen immediately after creating a clustered index; cluster ratios are 1.00000 or .999997, for example. Rows on the data pages are stored the same order as the rows in the index. The number of logical and physical I/Os needed for the data pages is (basically) the number of rows to be returned, divided by the number of rows per page. For a table with 10 rows per page, a query that needs to return 500 rows needs to read 50 pages if the data row cluster ratio is 1.

When the Data Row Cluster Ratio Is Low

When the data row cluster ratio is extremely low, the data rows are scattered on data pages with no relationship to the ordering of the keys. Nonclustered indexes often have low data row cluster ratios, since there is no relationship between the ordering of the index keys and the ordering of the data rows on data pages. When the data row cluster ratio is 0, or close to 0, the number of physical and logical I/Os required could be as much as 1 data page I/O for each row to be returned. A query that needs to return 500 rows needs to read 500 pages, or nearly 500 pages, if the data row cluster ratio is near 0 and the rows are widely scattered on the data pages. In a huge table, this still provides good performance, but in a table with less than 500 pages, the optimizer chooses the cheaper alternative – a table scan.

The size of the data cache is also used in calculating the physical I/O. If the data row cluster ratio is very low, and the cache is small, pages may be flushed from cache before they can be reused. If the cache is large, the optimizer estimates that some pages will be found in cache.

Result Set Size and Index Use

A range query that returns a small number of rows performs well with the index, however, range queries that return a large number of rows may not use the index—it may be more expensive to perform the logical and physical I/O for a large number of index pages plus a large number of data pages. The lower the data row cluster ratio, the more expensive it is to use the index.

At the leaf level of a nonclustered index or a clustered index on a data-only-locked table, the keys are stored sequentially. For a search argument on a value that matches 100 rows, the rows on the index leaf level fit on perhaps one or two index pages. The actual data rows might all be on different data pages. The following queries show how different data row cluster ratios affect I/O estimates. The *authors* table uses datarows locking, and has these indexes:

- A clustered index on *au_lname*
- A nonclustered index on *state*

Each of these queries returns about 100 rows:

```
select au_lname, phone
from authors
where au_lname like "E%"

select au_id, au_lname, phone
from authors
where state = "NC"
```

The following table shows the data row cluster ratio for each index, and the optimizer's estimate of the number of rows to be returned and the number of pages required.

SARG on	Data row cluster ratio	Row estimate	Page estimate	Data I/O size
au_lname	.999789	101	8	16K
state	.232539	103	83	2K

The basic information on the table is:

- The table has 262 pages.
- There are 19 rows per data page in the table.

While each of the queries has its search clauses in valid search-argument form, and each of the clauses matches an index, only the first query uses the index: for the other query, a table scan is cheaper than using the index. With 262 pages, the cost of the table scan is:

```
Table scan cost =  (262 /8) = 37 * 18   =        666
                 + 262 * 2              =        524
                                                ____
                                                1190
```

A Closer Look at the SARG Costing

Looking more closely at the tables, cluster ratios, and search arguments explains why the table scan is chosen:

- The estimate for the clustered index on *au_lname* includes just 8 physical I/Os:
  - 6 I/Os (using 16K I/O) on the data pages, because the data row cluster ratio indicates very high clustering.
  - 2 I/Os for the index pages (there are 128 rows per leaf page); 16K I/O is also used for the index leaf pages.

- The query using the search argument on *state* has to read many more data pages, since the data row cluster ratio is low. The optimizer chooses 2K I/O on the data pages. 83 physical I/Os is more than double the physical I/O required for a table scan (using 16K I/O).

Costing for Noncovering Index Scans

The basic formula for estimating I/O for queries accessing the data through a noncovering index is:

Leaf pages = Number of qualified rows / Leaf level rows per page

Data pages = Number of qualifying rows * Data row cluster adjustment

Scan cost = Number of nonleaf index levels * 18
+ (Leaf pages / Pages per IO) * Data page cluster adjustment * 18
+ (Data pages / Pages per IO) * Data page cluster adjustment * 18
+ Number of nonleaf index levels * 18
+ Leaf pages * 2
+ Number of qualifying rows * Data row cluster adjustment * 2

Added Cost for Forwarded Rows

If a data-only-locked table has forwarded rows, the cost of the extra I/O for accessing forwarded rows is added for noncovered index scans. The cost is computed by multiplying the number of forwarded rows in the table and the percent of the rows from the table that to be returned by the query. The added cost is:

Forwarded row cost = % of rows returned * Number of forwarded rows in the table

Query Costing for Queries Using *order by*

Queries that perform sorts for **order by** may create and sort, or they may be able to use the index to return rows by relying on the index ordering. For example, the optimizer chooses one of these access methods for a query with an **order by** clause:

- With no useful search arguments – Use a table scan, followed by sorting the worktable.

- With selective search argument or join on an index that does not match the **order by** clause – Use an index scan, followed by sorting the worktable.
- With a search argument or join on an index that matches the **order by** clause – An index scan using this index, with no worktable or sort.

Sorts are always 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_lname*, and the **order by** clause also includes the *au_id*, the query requires a sort.

If there are search arguments on indexes that match the **order by** clause, and other search arguments on indexes that do not support the required ordering, the optimizer costs both access methods. If the worktable and sort is required, the cost of performing the I/O for these operations is added to the cost of the index scan. If an index is potentially useful to help avoid the sort, **dbcc traceon(302)** prints a message while the search or join argument costing takes place. See “Sort Avert Messages” on page 18-9 for more information.

Besides the availability of indexes, two major factors determine whether the index is considered:

- The **order by** clause must specify a prefix subset of the index keys.
- The **order by** clause and the index must have compatible ascending/descending key ordering.

Prefix Subset and Sorts

For a query to use an index to avoid a sort step, the keys specified in the **order by** clause must be a prefix subset of the index keys. For example, if the index specifies the keys as A, B, C, D:

- The following **order by** clauses can use the index:
  - A
  - A, B
  - A, B, C
  - A, B, C, D
- And other set of columns cannot use the index. For example, these are NOT prefix subsets:
  - A, C
  - B, C, D

Key Ordering and Sorts

Both **order by** clauses and commands that create indexes can use the **asc** or **desc** (ascending or descending) ordering qualifications:

- For index creation, the **asc** and **desc** qualifications specify the order in which keys are to be stored in the index.
- In the **order by** clause, the ordering qualifications specify the order in which the columns are to be returned in the output.

To avoid a sort when using a specific index, the **asc** or **desc** qualifications in the **order by** clause must either be exactly the same as those used to create the index, or must be exactly the opposite.

Specifying Ascending or Descending Order for Index Keys

Queries that use a mix of ascending and descending order in an **order by** clause do not perform a separate sort step if the index was created using the same mix of ascending and descending order as that specified in the **order by** clause, or if the index order is the reverse of the order specified in the **order by** clause. Indexes are scanned forward or backward, following the page chain pointers at the leaf level of the index.

For example, this command creates an index on the *titles* table with *pub_id* ascending and *pubdate* descending:

```
create index pub_ix
    on titles (pub_id asc, pubdate desc)
```

The rows are ordered on the pages as shown in Figure 6-5. When the ascending and descending order in the query matches the index creation order, the result is a forward scan, starting at the beginning of the index or at the first qualifying row, returning the rows in order from each page, and following the next-page pointers to read subsequent pages.

If the ordering in the query is the exact opposite of the index creation order, the result is a backward scan, starting at the last page of the index or the page containing the last qualifying row, returning rows in backward order from each page, and following previous page pointers.

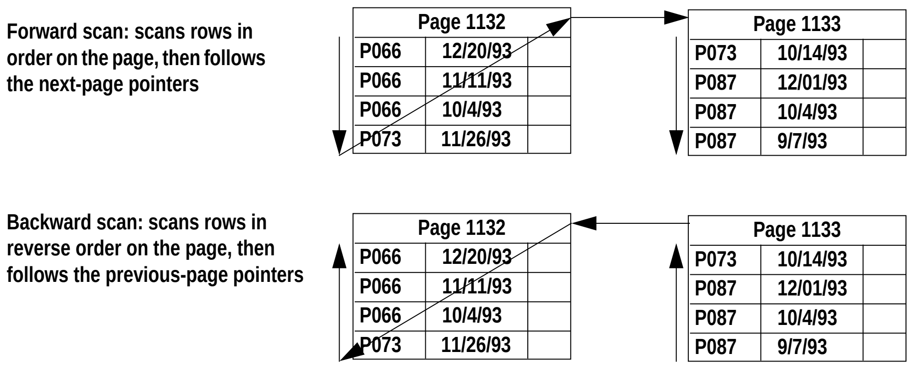


Figure 6-5: Forward and backward scans on an index

The following query using the index shown in Figure 6-5 performs a forward scan:

```
select *
from titles
order by pub_id asc, pubdate desc
```

This query using the index shown in Figure 6-5 performs a backward scan:

```
select *
from titles
order by pub_id desc, pubdate asc
```

For the following two queries on the same table, the plan requires a sort step, since the **order by** clauses do not match the ordering specified for the index:

```
select *
from titles
order by pub_id desc, pubdate desc
```

```
select *
from titles
order by pub_id asc, pubdate asc
```

➤ ***Note***

Parallel sort operations are optimized very differently for partitioned tables. See Chapter 13, “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. However, when comparing indexes that are very selective, but not ordered, versus indexes that are ordered, but not selective:

- Access costs are low for the more selective index, and so are sort costs.
- Access costs are high for the less selective index, and may exceed the cost of access using the more selective index and sort.

Allpages-Locked Tables with Clustered Indexes

For allpages-locked tables with clustered indexes, **order by** queries that match the index keys are efficient:

- If there is also a search argument that uses the index, the index key positions the search on the data page for first qualifying row.
- The scan follows the next-page pointers until all qualifying rows have been found.
- No sort is needed.

In Figure 6-6, the index was created in ascending order, and the **order by** clause does not specify the order, so ascending is used by default.

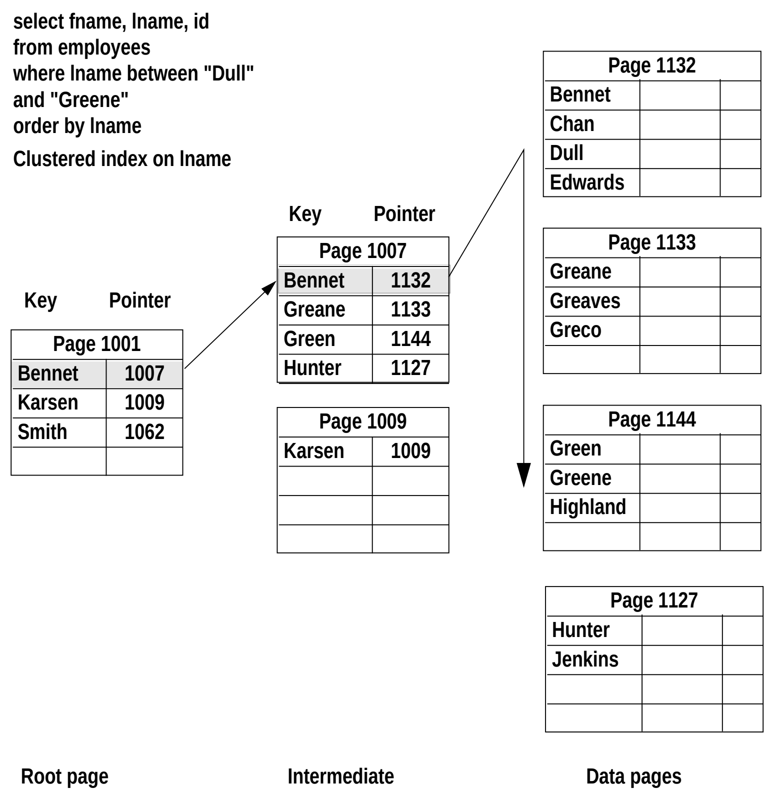


Figure 6-6: An order by query using a clustered index, allpages locking

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 6-7.

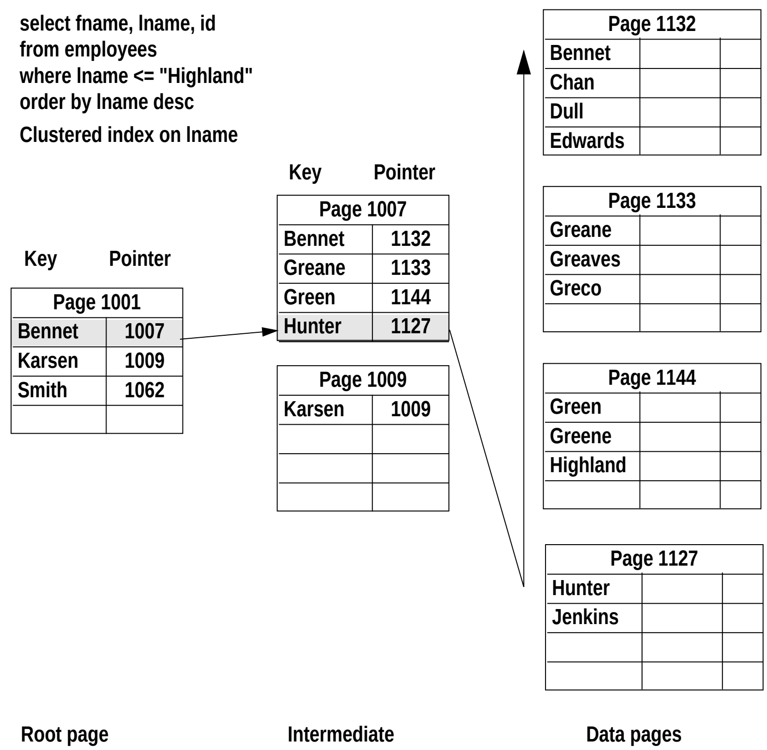


Figure 6-7: An order by desc query using a clustered index

Sorts When the Index Covers the Query

When an index covers the query and 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
```

Sorts and Noncovering Indexes

With a noncovering index, Adaptive Server determines whether using the index that supports the ordering requirements is cheaper than performing a table scan or using a more selective index, and then inserting rows into a worktable and sorting the data. The cost of using the index depends on the number of rows and the data row cluster ratio.

Backward 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 and indexes involved can be scanned with a backward scan 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.

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 17-116 of the *System Administration Guide* for more information on this parameter. Also, see "Index Scans" on page 39-60 for information on the number of ascending and descending scans performed and "Deadlocks by Lock Type" on page 39-67 for information on detecting deadlocks.

Access Methods and Costing for *or* and *in* Clauses

When a query on a single table contains **or** clauses or an **in (*values_list*)** clause, it can be optimized in different ways, depending on the presence of indexes, the selectivity of the search arguments, the existence of other search arguments, and whether or not the clauses might return duplicate rows.

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

Preprocessing 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"
```

Methods for Processing *or* Clauses

A single-table query including **or** clauses is a union of more than one query. Although some rows may match more than one of the conditions, each row must be returned only once. Depending on indexes and query clauses, **or** queries can be resolved by one of these methods:

- If any of the clauses linked by **or** is not indexed, the query must use a table scan. If there is an index on *type*, but no index on *advance*, this query performs a table scan:

```
select title_id, price
from titles
where type = "business" or advance > 10000
```

- If there is a possibility that one or more of the **or** clauses could match values in the same row, the query is resolved using the **OR strategy**, also known as using a **dynamic index**. The OR strategy selects the row IDs for matching rows into a worktable, and sorts the worktable to remove duplicate row IDs. For example, there can be rows for which both of these conditions are true:

```
select title_id
from titles
where pub_id = "P076" or type > "business"
```

If there is an index on *pub_id*, and another on *type*, the OR strategy can be used. See “Dynamic Index (OR Strategy)” on page 6-26 for more information.

- If there is no possibility that the or clauses can select the same row, the query can be resolved with multiple matching index scans, also known as the **special OR strategy**. The special OR strategy does not require a worktable and sort. The or clauses in this query cannot select the same row twice:

```
select title_id, price
from titles
where pub_id = "P076" or pub_id = "P087"
```

With an index on *pub_id*, this query can be resolved using two matching index scans. See “Multiple Matching Index Scans (Special OR Strategy)” on page 6-28 for more information.

- The costs of index access for each or clause are added together, and the cost of the sort, if required. If sum of these costs is greater than a table scan, the table scan is chosen. For example, this query uses a table scan if the total cost of all of the indexed scans on *pub_id* is greater than the table scan:

```
select title_id, price
from titles
where pub_id in ("P095", "P099", "P128", "P220", ,
"P411", "P445", "P580", "P988")
```

- If the query contains additional search arguments on indexed columns, predicate transformation may add optimizable search arguments, adding alternative optimization options. The cost of using all alternative access methods is compared, and the cheapest alternative is selected. This query contains a search argument on *type* as well as clauses linked with or:

```
select title_id, type, price from titles
where type = "business"
and (pub_id = "P076" or pubdate > "12/1/93")
```

With a separate index on each search argument, the optimizer uses the least expensive access method:

  - The index on *type*

- The OR strategy on *pub_id* and *pubdate*

When Table Scans Are Used for *or* Queries

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 columns is not indexed, so the only way to resolve the query conditions is to perform a table scan.

Dynamic Index (OR Strategy)

If the query uses the OR strategy because the query could return duplicate rows, the appropriate indexes are used to retrieve the row IDs for rows that satisfy each **or** clause. The row IDs for each or clause are stored in a worktable. Since the worktable contains only row IDs, it is called a "dynamic index." Adaptive Server then sorts the worktable to remove the duplicate row IDs. The row IDs are used to retrieve the rows from the base tables. The total cost of the query includes:

- The sum of the index accesses, that is, for each **or** clause, the cost of using the index to access the row IDs on the leaf pages of the index (or on the data pages, for a clustered index on an allpages-locked table)
- The cost of reading the worktable and performing the sort
- The cost of using the row IDs to access the data pages

Figure 6-8 illustrates the process of building and sorting a dynamic index for an **or** query on two different columns.

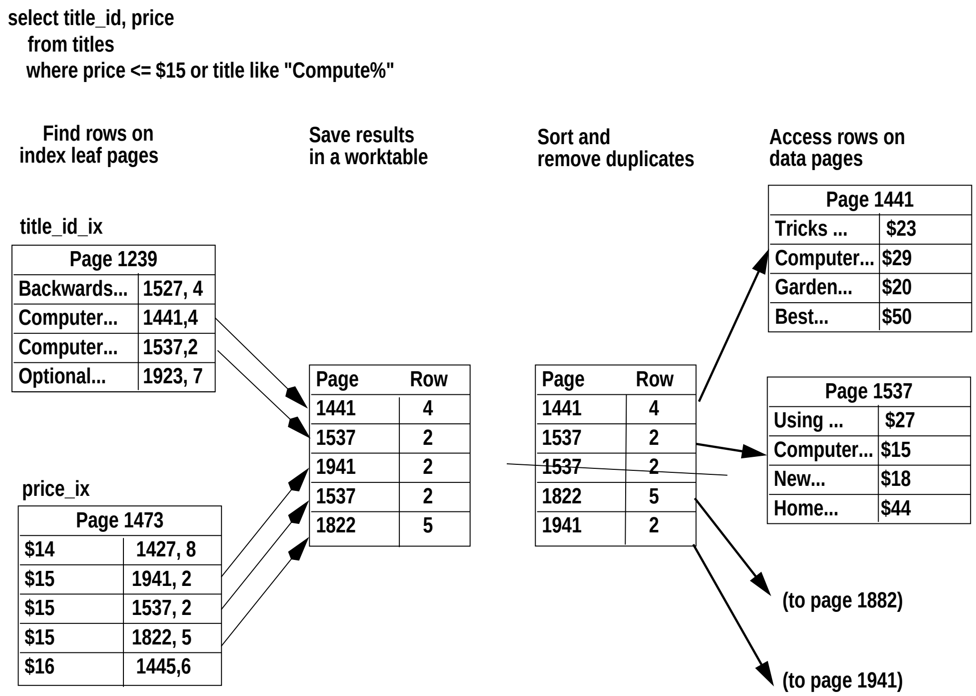


Figure 6-8: Resolving or queries using the OR strategy

As shown in Figure 6-8, the optimizer can choose to use a different index for each clause.

showplan displays “Using Dynamic Index” and “Positioning by Row IDentifier (RID).” when the OR strategy is used. See “Dynamic Index Message (OR Strategy)” on page 17-36 for more information.

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 during queries that use the OR strategy depends on the locking scheme of the table. See “Locking During or Processing” on page 26-28.

Multiple Matching Index Scans (Special OR Strategy)

Adaptive Server uses multiple matching index scans when the **or** clauses are on the same table, and there is no possibility that the **or** clauses will return duplicate rows. For example, 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, using the index on *title_id*. The total cost of the query is the sum of the multiple index accesses performed. If the index on *title_id* has 3 levels, each **or** clause requires 3 index reads, plus one data page read, so the total cost for each clause is 4 logical and 4 physical I/Os, and the total query cost is estimated to be 12 logical and 12 physical I/Os.

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. **showplan** displays the message “Using *N* Matching Index Scans” when the special OR strategy is used. See “Matching Index Scans Message” on page 17-35.

How Aggregates Are Optimized

Aggregates are processed in two steps:

- First, appropriate indexes are used to retrieve the appropriate rows, or a table scan is performed. 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.

Vector aggregates can use a covering composite 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 by scanning the leaf level of the nonclustered index:

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

Scalar aggregates can also use covering indexes to reduce I/O. For example, the following query can use the index on *type, price*:

```
select min(price)
    from titles
```

Table 6-1 shows some of the 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 6-1: Special access methods for aggregates

Aggregate	Index Description	Access Method
min	Scalar aggregate is leading column	Use first the value on the root page of the index.
max	Clustered index on an allpages-locked table	Follow the last pointer on root page and intermediate pages to data page, and return the last value.
	Clustered index on a data-only-locked table Any nonclustered index	Follow last pointer on root page and intermediate pages to leaf page, and return the last value.
count(*)	Nonclustered index or clustered index on a data-only-locked table	Count all rows in the leaf level of the index with the smallest number of pages.
count(*col_name*)	Covering nonclustered index, or covering clustered index on data-only-locked table	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 leading index 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.

However, when **min** and **max** are used together, this optimization is not available. The entire leaf level of an index is scanned to locate the first and last values.

min and **max** optimizations are not applied if:

- The expression inside the **max** or **min** function is anything but a column. When *numeric_col* has a nonclustered index:

- **max(*numeric_col**2)** contains an operation on a column, so the query performs a leaf-level scan of the index.
- **max(*numeric_col*)*2** uses **max** optimization, because the multiplication is performed on the result of the function.

- There is another aggregate in the query.
- There is a **group by** clause.

Queries That Use Both *min* and *max*

If you have optimizable **max** and **min** aggregates, you should get much better performance by putting them in separate queries. For example, even if there is an index with *price* as the leading key, this query results in a full leaf-level scan of the index:

```
select max(price), min(price)
    from titles
```

When you separate them, Adaptive Server uses the index once for each of the two queries, rather than scanning the entire leaf level. This examples shows two queries:

```
select max(price)
    from titles
select min(price)
    from titles
```

How Update Operations Are Performed

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:

- 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 cannot change its length.
- The column being updated cannot be the key, or part of the key, of a clustered index on an allpages-locked table. Because the rows in a clustered index on an allpages-locked table 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 satisfies the conditions listed in “Restrictions on Update Modes Through Joins” on page 6-36.
- 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).

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 change 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 satisfies the conditions listed in "Restrictions on Update Modes Through Joins" on page 6-36.
- The affected columns are not used for referential integrity.

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 index, Adaptive Server uses the index to determine the new location for the row; otherwise, Adaptive Server inserts the new row at the end of the heap.

For an expensive direct update, the following requirements must be met:

- The length of a data row is changed so that the row no longer fits on the same data page, and the row is moved to a different page, or the update affects key columns for the clustered index.

- The index used to find the row is not changed by the update.
- The update statement satisfies the conditions listed in "Restrictions on Update Modes Through Joins" on page 6-36.
- The affected columns are not used for referential integrity.

An expensive direct update is the slowest type of direct update. The delete is performed on one data page, and the insert is performed on a different data page. All index entries must be updated, since the row location is changed.

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.
- Reads the log records a second time, and performs all inserts on the data pages, and inserts any affected index rows.

When Deferred Updates Are Required

Deferred updates are always required for:

- Updates that use self-joins
- Updates to columns used for self-referential integrity
- Updates to a table referenced in a correlated subquery

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 reread 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 to access the table 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 6-9 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 6-9 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.

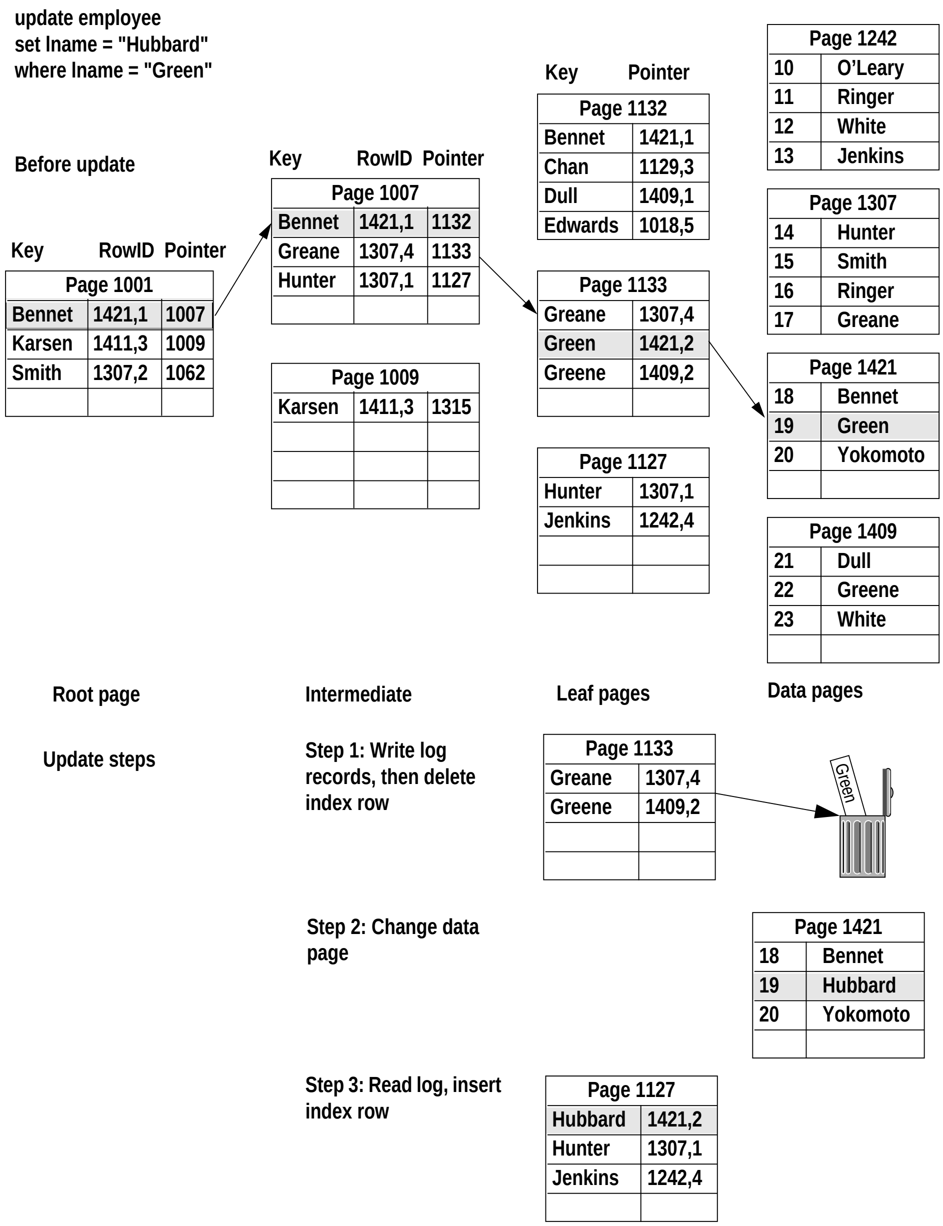


Figure 6-9: Deferred index update

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!

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.

Restrictions on Update Modes Through Joins

Updates and deletes that involve joins can be performed in direct, deferred_varcol, or deferred_index mode when the table being updated is the outermost table in the join order, or when it is preceded in the join **order by** tables where only a single row qualifies.

Joins and Subqueries in Update and Delete Statements

The use of the **from** clause to perform joins in **update** and **delete** statements is a Transact-SQL extension to ANSI SQL. Subqueries in ANSI SQL form can be used in place of joins for some updates and deletes.

This example uses the **from** syntax to perform a join:

```
update t1 set t1.c1 = t1.c1 + 50
from t1, t2
where t1.c1 = t2.c1
and t2.c2 = 1
```

The following example shows the equivalent update using a subquery:

```
update t1 set c1 = c1 + 50
where t1.c1 in (select t2.c1
                from t2
                where t2.c2 = 1)
```

The update mode that is used for the join query depends on whether the updated table is the outermost query in the join order—if it is not the outermost table, the update is performed in deferred mode. The update that uses a subquery is always performed as a direct, deferred_varcol, or deferred_index update.

For a query that uses the **from** syntax and performs a deferred update due to the join order, use **showplan** and **statistics io** to determine whether rewriting the query using a subquery can improve performance. Note that not all queries using **from** can be rewritten to use subqueries.

Deletes and Updates in Triggers vs. Referential Integrity

Triggers that join user tables with the *deleted* or *inserted* tables are run in deferred mode. If you are using triggers solely to implement referential integrity, and not to cascade updates and deletes, then using declarative referential integrity in place of triggers may avoid the penalty of deferred updates in triggers.

Optimizing Updates

showplan messages provide information about whether an update is 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.

These types of direct updates depend on information that is available only at run time, since the page actually has to be fetched and examined in order to determine whether the row fits on the page.

Designing for Direct Updates

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 6-2 shows how indexes affect the 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:

- Update of a variable-length key column:

```
update titles set title_id = value
    where title_id = "T1234"
```

- Update of a fixed-length nonkey column:

```
update titles set date = value
        where title_id = "T1234"
```

- Update of a variable-length nonkey column:

```
    update titles set notes = value
        where title_id = "T1234"
```

Table 6-2 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 6-2: 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 an index 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 “Using max_rows_per_page on Allpages-Locked Tables” on page 31-21.

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 "Transaction Detail" on page 39-42 for more information.

7 Access Methods and Query Costing for Joins and Subqueries

This chapter introduces the methods that Adaptive Server uses to access rows in tables when more than one table is used in a query, and how the optimizer costs access.

This chapter contains the following sections:

In determining the cost of multitable queries, Adaptive Server uses many of the same formulas discussed in Chapter 6, "Access Methods and Query Costing for Single Tables."

Costing and Optimizing Joins

Joins extract information from two or more tables. In a two-table join, one table is treated as the outer table and the other table is treated as 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.

Optimizing join queries is extremely important for system performance, since relational databases make heavy use of joins. Queries that perform joins on several tables are especially critical to performance, as explained in the following sections.

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 17-4 for an example that joins three tables. Some subqueries are also converted to joins. See "Flattening in, any, and exists Subqueries" on page 7-23.

Join Processing

By default, Adaptive Server uses nested-loop joins, and also considered merge joins, if this feature is enabled at the server-wide or session level.

When merge joins are enabled, Adaptive Server can use either nested-loop joins or merge joins to process queries involving two or more tables. For each join, the optimizer costs both methods. For queries involving more than two tabes, optimizer examines query costs for merge joins and for nested-loops, and chooses the mix of merge and nested-loop joins that provides the cheapest query cost.

Index Density and Joins

The optimizer uses a statistic called the **total density** to estimate the number of rows in a joined table that match a particular value during the join. See “Density Values and Joins” on page 5-20 for more information.

The query optimizer uses the total density to estimate the number of rows that will be returned for each scan of the inner table of a join. For example, if the optimizer is considering a nested-loop join with a 250,000-row table, and the table has a density of .0001, the optimizer estimates that an average of 25 rows from the inner table match for each row that qualifies in the outer table.

optdiag reports the total density for each column for which statistics have been created. You can also see the total density used for joins in **dbcc traceon(302)** output.

Multicolumn Densities

Adaptive Server maintains the total density for each prefix subset of columns in a composite index. If two tables are being joined on multiple leading columns of a composite 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.

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 converted to the other, and an index can only be used for one side of the join. See "Datatype Mismatches and Query Optimization" on page 5-21 for more information.

Join Permutations

When you are joining four or fewer tables, Adaptive Server considers all possible permutations of join orders for the tables. However, due to the iterative nature of Adaptive Server's optimizer, queries on more than four tables examine join order combinations in sets of two to four tables at a time. This grouping during join order costing is used because the number of permutations of join orders multiplies with each additional table, requiring lengthy computation time for large joins. 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.

If the number of tables in a join is greater than 25, Adaptive Server automatically reduces the number of tables considered at a time. Table 7-1 shows the default values.

Table 7-1: Tables considered at a time during a join

Tables Joined	Tables Considered at a Time
4 – 25	4
26 – 37	3
38 – 50	2

The optimizer starts by considering the first two to four tables, and determining the best join order for those tables. It remembers the outer table from the best plan involving the tables it examined and eliminates that table from the set of tables. Then, it optimizes the best set of tables out of the remaining tables. It continues until only two to four tables remain, at which point it optimizes them.

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 determined to be:

```
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, 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 4 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
```

Outer Joins and Join Permutations

Outer joins restrict the set of possible join orders. When the inner member of an outer join is compared to an outer member, the outer member must precede the inner member in the join order. The only join permutations that are considered for outer joins are those that meet this requirement. For example, these two queries perform outer joins, the first using ANSI SQL syntax, the second using Transact-SQL syntax:

```
select T1.c1, T2.c1, T3.c2, T4.c2
from T4 inner join T1 on T1.c1 = T4.c1
left outer join T2 on T1.c1 = T2.c1
left outer join T3 on T2.c2 = T3.c2
```

```
select T1.c1, T2.c1, T3.c2, T4.c2
from T1 , T2, T3, T4
where T1.c1 *= T2.c1
and T2.c2 *= T3.c2
and T1.c1 = T4.c1
```

The only join orders considered place *T1* outer to *T2* and *T2* outer to *T3*. The join orders considered by the optimizer are:

```
T1, T2, T3, T4
T1, T2, T4, T3
T1, T4, T2, T3
T4, T1, T2, T3
```

Nested-Loop Joins

Nested-loop joins provide efficient access when tables are indexed on join columns. The process of creating the result set for a nested-loop 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 7-1.

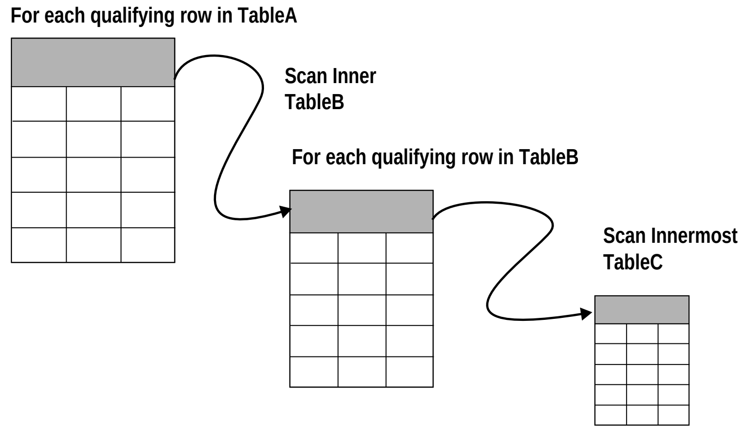


Figure 7-1: Nesting of tables during a nested-loop join

In Figure 7-1, the access to the tables to be joined is nested:

- *TableA* is accessed once. If the table has no useful indexes, a table scan is performed. If an index can reduce I/O costs, the index is used to locate the rows.
- *TableB* is accessed once for each qualifying row in *TableA*. If 15 rows from *TableA* match the conditions in the query, *TableB* is accessed 15 times. If *TableB* has a useful index on the join column, it might require 3 I/Os to read the data page for each scan, plus one I/O for each data page. The cost of accessing TableB would be 60 logical I/Os.
- *TableC* is accessed once for each qualifying row in *TableB* **each time** that *TableB* is accessed. If 10 rows from *TableB* match for each row in *TableA*, then *TableC* is scanned 150 times. If each access to TableC requires 3 I/Os to locate the data row, the cost of accessing TableC is 450 logical I/Os.

If *TableC* is small, or has a useful index, the I/O count stays reasonably small. If *TableC* is large and has no useful index on the join columns, the optimizer may choose to use a sort-merge join or the reformatting strategy to avoid performing extensive I/O.

Cost Formula for Nested-Loop Joins

For a nested-loop join with two tables, the formula for estimating the cost is:

```
Join cost = Cost of accessing A +
        # of qualifying rows in A * Pages of B to scan for each qualifying row
```

With additional tables, the cost of a nested-loop join is:

```
     Cost of accessing outer table
+    (Number of qualified rows in outer) * ( Cost of accessing inner table)
+    ...
+    (Number of qualified rows from previous) * (Cost of accessing innermost table)
```

Choice of Inner and Outer Tables for Nested-Loop Joins

The outer table is usually the one that has:

- The smallest number of qualifying rows, and / or
- The largest numbers of I / Os 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, so that the large table is scanned only once.
- The indexed table as the inner table, so that each time the inner table is accessed, it takes only a few reads to find rows.

Access Methods and Costing for Sort-Merge Joins

There are four possible execution methods for merge joins:

- Full-merge join – The two tables being joined have useful indexes on the join columns. The tables do not need to be sorted, but can be merged using the indexes.

- Left-merge join – Sort the inner table in the join order, then merge with the left, outer table.
- Right-merge join – Sort the outer table in the join order, then merge with the right, inner table.
- Sort-merge join – Sort both tables, then merge.

Merge joins always operate on stored tables, either user tables or worktables created for the merge join. When a worktable is required for a merge join, it is sorted into order on the join key, then the merge step is performed. The costing for any merge joins that involve sorting includes the estimated I/O cost of creating and sorting a worktable. For full-merge joins, the only cost involved is scanning the tables.

Figure 7-2 provides diagrams of the merge join types.

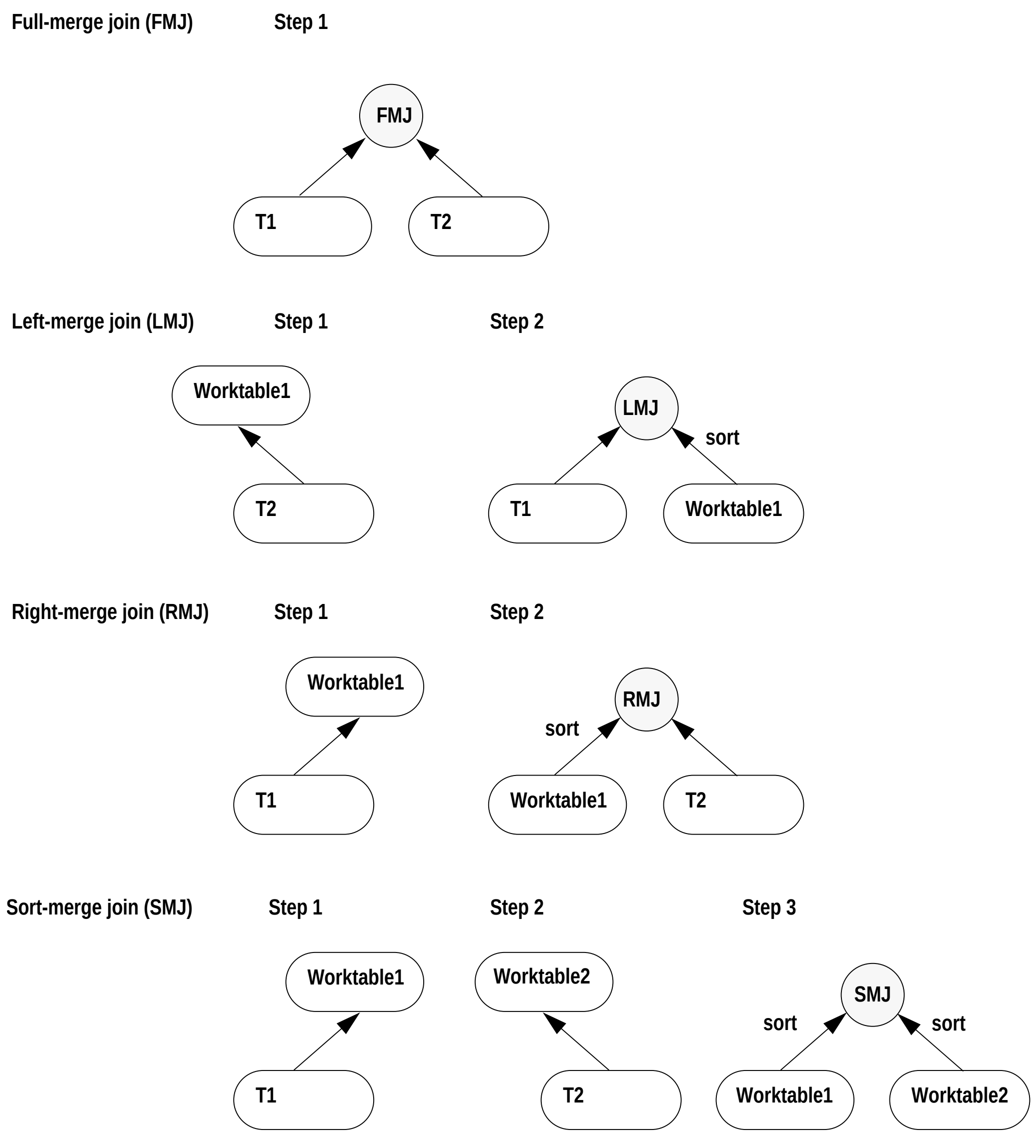


Figure 7-2: Merge join types

How a Full-Merge Join Is Performed

If both *Table1* and *Table2* have indexes on the join key, this query can use a full-merge join:

```
select *
    from Table1, Table2
    where Table1.c1 = Table2.c2
    and Table1.c1 between 100 and 120
```

If both tables are allpages-locked tables with clustered indexes, and *Table1* is chosen as the outer table, the index is used to position the search on the data page at the row where the value equals 100. The index on *Table2* is also used to position the scan at the first row in *Table2* where the join column equals 100. From this point, rows from both tables are returned as the scan moves forward on the data pages.

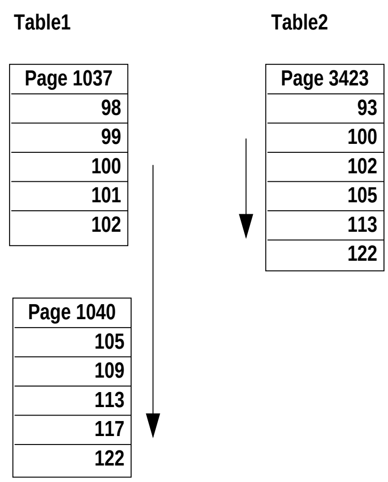


Figure 7-3: A serial merge scan on two tables with clustered indexes

Merge joins can also be performed using nonclustered indexes. The index is used to position the scan on the first matching value on the leaf page of the index. For each matching row, the index pointers are used to access the data pages. Figure 7-4 shows a full-merge scan using a nonclustered index on the inner table.

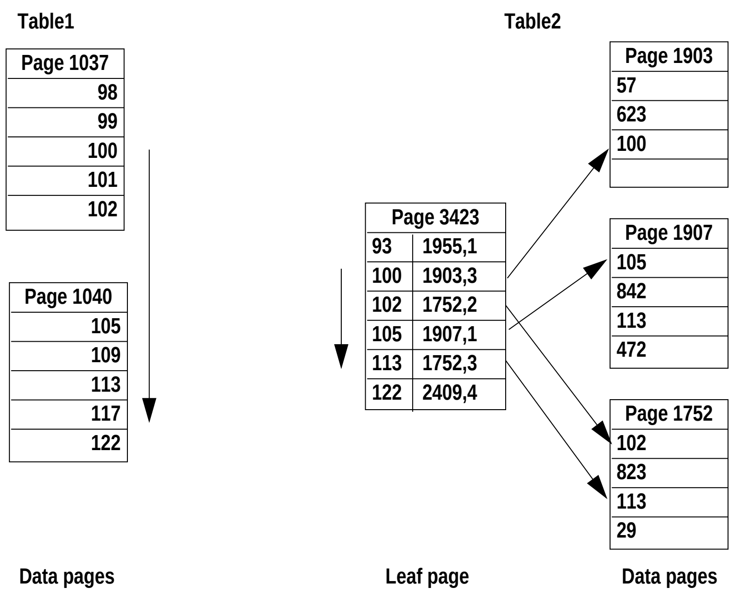


Figure 7-4: Full merge scan using a nonclustered index on the inner table

How a Right-Merge or Left-Merge Join Is Performed

A right-merge or left-merge join always operates on a user table and a worktable created for the merge join. There are two steps:

1. A table or set of tables is scanned, and the results are inserted into a worktable.
2. The worktable is sorted and then merged with the other table in the join, using the index.

How a Sort-Merge Join Is Performed

For a sort-merge join, there are three steps, since the inputs to the sort-merge joins are both sorted worktables:

1. A table or set of tables is scanned and the results are inserted into one worktable. This will be the outer table in the merge.
2. Another table is scanned and the results are inserted into another worktable. This will be the inner table in the merge.

3. Each of the worktables is sorted, then the two sorted result sets are merged.

Example of Mixed Joins

This query performs a mixture of merge and nested-loop joins:

```
select pub_name, au_lname, price
from titles t, authors a, titleauthor ta,
            publishers p
where t.title_id = ta.title_id
    and a.au_id = ta.au_id
    and p.pub_id = t.pub_id
    and type = 'business'
    and price < $25
```

Adaptive Server executes this query in three steps:

- Step 1 uses 3 worker processes to scan *titles* as the outer table, performing a full-merge join with *titleauthor* and then a nested-loop join with *authors*. No sorting is required for the full-merge join. *titles* has a clustered index on *title_id*. The index on *titleauthor*, *ta_ix*, contains the *title_id* and *au_id*, so the index covers the query. The results are stored in *Worktable1*, for use in the sort-merge join performed in Step 3.
- Step 2 scans the *publishers* table, and saves the needed columns (*pub_name* and *pub_id*) in *Worktable2*.
- In Step 3:
    - *Worktable1* is sorted into join column order, on *pub_id*.
    - *Worktable2* is sorted into order on *pub_id*.
    - The sorted results are merged.

Figure 7-5 shows the steps.

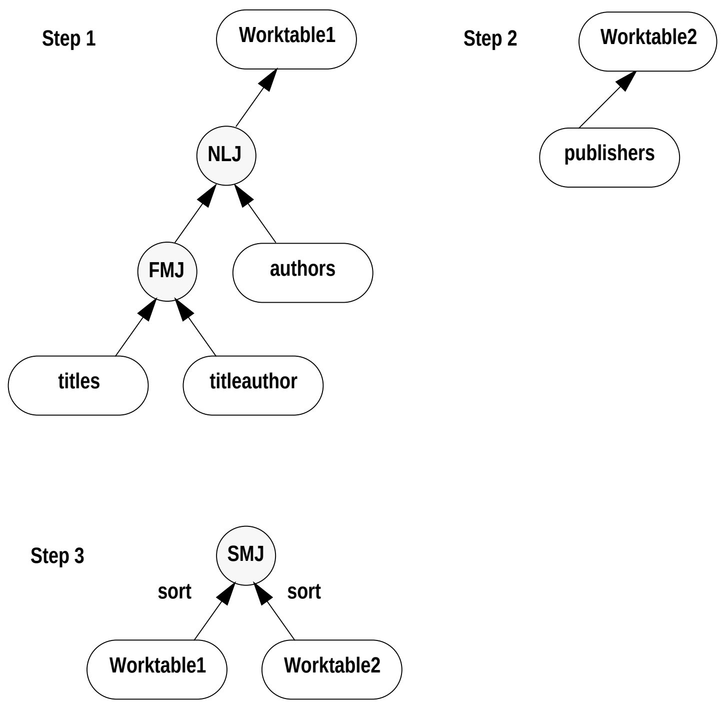


Figure 7-5: Multiple steps in processing a merge join

showplan Messages for Sort-Merge Joins

showplan messages for each type of merge join appear as specific combinations:

- Full-merge join – There are no "FROM TABLE Worktable" messages, only the "inner table" and "outer table" messages for base tables in the query.
- Right-merge join – The "outer table" is always a worktable.
- Left-merge join – The "inner table" is always a worktable.
- Sort-merge join – Both tables are worktables.

For more information, see "showplan Messages Describing Access Methods, Caching, and I/O Cost" on page 17-22.

Costing for Merge Joins

The total cost for merge joins depends on:

- The type of merge join.
  - Full-merge joins do not require sorts and worktables.
  - For right-merge and left-merge joins, one side of the join is selected into a worktable, then sorted.
  - For sort-merge joins, both sides of the join are selected into worktables, and each worktable is sorted.
- The type of index used to scan the tables while performing the merge step.
- The locking scheme of the underlying table: costing models for most scans are different for allpages locking than data-only locking. Clustered index access cost on data-only-locked tables is more comparable to nonclustered access.
- Whether the query is executed in serial or parallel mode.
- Whether the outer table has duplicate values for the join key.

In general, when comparing costs between a nested-loop join and a merge join for the same tables, using the same indexes, the cost for the outer table remains the same. Access to the inner table costs less for a merge join because the scan remains positioned on the leaf pages as matching values are returned, saving the logical I/O cost of scanning down the index from the root page each time.

Costing for a Full-Merge Join with Unique Values

If a full-merge join is performed in serial mode and there is no need to sort the tables, the cost of a merge join on *T1* and *T2* is the sum of the cost of the scans of both tables, as long as all join values are unique:

Merge join cost = Cost of scan of T1 + Cost of scan of T2

The cost saving of a merge join over a nested-loop join is:

- For a nested-loop join, access to the inner table of the join starts at the root page of the index for each row from the outer table that qualifies.
- For a full-merge join, the upper levels of the index are used for the first access, to position the scan:

- On the leaf page of the index, for nonclustered indexes and clustered indexes on data-only-locked tables
- On the data page, if there is a clustered index on an allpages-locked table

The higher levels of the index do not need to be read for each matching outer row.

Example: Allpages-Locked Tables with Clustered Indexes

For allpages-locked tables where clustered indexes are used to perform the scans, the search arguments on the index are used to position the search on the first matching row of each table. The total cost of the query is the cost of scanning forward on the data pages of each table. For example, with clustered indexes on *t1(c1)* and *t2(c1)*, the query on two allpages-locked tables can use a full-merge join:

```
select t1.c2, t2.c2
from t1, t2
where t1.c1 = t2.c1
and t1.c1 >= 1000 and t1.c1 < 1100
```

If there are 100 rows that qualify from *t1*, and 100 rows from *t2*, and each of these tables has 10 rows per page, and an index height of 3, the costs are:

- 3 index pages to position the scan on the first matching row of *t1*
- Scanning 10 pages of *t1*
- 3 index pages to position the scan on the first matching row of *t2*
- Scanning 10 pages of *t2*

Costing for a Full-Merge Join with Duplicate Values

If the outer table in a merge join has duplicate values, the inner table must be accessed from the root page of the index for each duplicate value. This query is the same as the previous example:

```
select t1.c2, t2.c2
from t1, t2
where t1.c1 = t2.c1
and t1.c1 >= 1000 and t1.c1 < 1100
```

If *t1* is the outer table, and there are duplicate values for some of the rows in *t1*, so that there are 120 rows between 1000 and 1100,with 20 duplicate values, then each time one of the duplicate values is accessed, the scan of *t2* is restarted from the root page of the index. If one row for *t2* matches each value from *t1*, the I/O costs for this query are:

- 3 index pages to position on the first matching row of *t1*
- Scanning 12 pages of *t1*
- 3 index pages to position on the first matching row of *t2*, plus an I/O to read the data page
- For the remaining rows:
  - If the value from *t1* is a duplicate, the scan of *t2* restarts from the root page of the index.
  - For all values of *t1* that are not duplicates, the scan remains positioned on the leaf level of *t2*. The scan on the inner table remains positioned on the leaf page as rows are returned until the next duplicate value in the outer table requires the scan to restart from the root page.

This formula gives the cost of the scan of the inner table for a merge join:

```
Cost of scan of inner = Num duplicate values * (index height + scan size)
                        + Num unique values * scan size
```

The *scan size* is the number of pages of the inner table that need to be read for each value in the outer table. For tables where multiple inner rows match, the scan size is the average number of pages that need to be read for each outer row.

Costing for Sorts

Sort cost during sort-merge joins depends on:

- The size of the worktables, which depends on the number of columns and rows selected
- The setting for the **number of sort buffers** configuration parameter, which determines how many pages of the cache can be used

These variables affect the number of merge runs required to sort the worktable.

Worktable Size for Sort-Merge Joins

When a worktable is created for a merge join that requires a sort, only the columns that are needed for the result set and for later joins in the query execution are selected into the worktable. When the worktable for the *titles* table is created for the join shown in Figure 7-5 on page 7-14:

- Worktable1 includes the *price* and *authors.state*, because they are part of the result set, and *pub_id*, because it is needed for a subsequent join.
- Worktable2 includes the *publishers.state* column because it is part of the result set, and the *pub_id*, because it is needed for the merge step.

The *type* column is used as a search argument while the rows from *titles* are selected, but since it is not used later in the query or in the result set, it is not included in the worktable.

Sort Buffers and Sorts for Merge Joins

Each sort performed for a merge join can use up to **number of sort buffers** for intermediate sort steps. Sort buffers for worktable sorts are allocated from the cache used by *tempdb*. If the number of pages to be sorted is less the **number of sort buffers**, then the number of buffers reserved for the sort is the number of pages in the worktable.

When Merge Joins Cannot Be Used

Merge joins are not used:

- For joins using <, >, <=, >=, or != on the join columns.
- For outer joins, that is, queries using ***=** or **=***, and **left join** and **right join**.
- For tables that include a *text* or *image* column or Java object columns in the select list or in a **where** clause.
- For subqueries that are not flattened or materialized in parallel queries.
- For multitable updates and deletes, such as:

```
update R set a = 5
    from R, S, T
    where ...
```

- For joins to perform referential integrity checks for **insert**, **update**, and **delete** commands. These joins are generated internally to check for the existence of the column values. They usually involve joins that return a single value from the referenced table.

Often, these joins are supported by indexes. There would be no benefit from using a merge join for constraint checks.

- When the number of bytes in a row for a worktable would exceed the page-size limit (1960 bytes of user data) or the limit on the number of columns (250). If the select list and required join columns for a join would create a worktable that exceeds either of these limits, the optimizer does not consider performing a merge join at that point in the query plan.
- When the use of worktables for a merge join would require more than the maximum allowable number of worktables for a query (14).

There are some limits on where merge joins can be used in the join order:

- Merge joins can be performed only before, never after, an existence join. Some **distinct** queries are turned into existence joins, and merge joins are not used for these.
- Full-merge joins and left-merge joins can be performed only on the outermost tables in the join order.

Use of Worker Processes for Merge Joins

When parallel processing is enabled, merge joins can use multiple worker processes to perform:

- The scan that selects rows into the worktables
- Worktable sort operations
- The merge join and subsequent joins in the step

See “Parallel Range-Based Scans” on page 12-12 for more information.

Recommendations for Improved Merge Performance

Here are some suggestions for improving sort-merge join performance:

- Select only needed columns for tables used in merge joins to reduce the size of worktables. Avoid using **select *** unless you need all columns of the tables. This reduces the load on *tempdb* and the cost of sorting the result tables.

- If you are concerned about possible performance impacts of merge joins or possible space problems in *tempdb*, see Chapter 21, “Introduction to Abstract Plans,” for a discussion of how abstract query plans can help determine which queries on your system use merge joins.
- Look for opportunities for index covering. One example is queries where joins are in the form:

```
select t1.c3, t3.c4
from t1, t2, t3
wehre t1.c1 = t2.c1 and t2.c2 = t3.c2
and ...
```

  and columns from *t2* are not in the select list, or only the join columns are in the select list. An index on the join columns, *t2(c1, c2)* covers the query, allowing a merge join to avoid accessing the data pages of *t2*.
- Merge joins can use indexes created in ascending or descending order when two tables are joined on multiple columns, such as these:

```
A.c1 = B.c1 and A.c2 = B.c2 and A.c3 = B.c3
```

  The column order specified for the indexes must be an exact match, or exactly the reverse, for all columns to be used as join predicates when costing the join and accessing the data. If there is a mismatch of ordering in second or subsequent columns, only the matching columns are used for the join, and the remaining columns are used to restrict the results after the row has been retrieved. This table shows some examples for the query above:

Index creation order	Clauses used as join predicates
A(c1 asc, c2 asc, c3 asc) B(c1 asc, c2 asc, c3 asc)	All three clauses.
A(c1 asc, c2 asc, c3 asc) B(c1 desc, c2 desc, c3 desc)	All three clauses.
A(c1 asc, c2 asc, c3 asc) B(c1 desc, c2 desc, c3 asc)	The first two join clauses are used as join predicates and the third clause is evaluated as a restriction on the result.
A1(c1 asc, c2 desc, c3 desc) B1(c1 desc, c2 desc, c3 asc)	Only the first join clause is used as a join predicate. The remaining two clauses is evaluated as restrictions on the result set.

Index key ordering is generally chosen to eliminate sort costs for **order by** queries. Using compatible ordering for frequently joined tables can also reduce join costs.

Enabling and Disabling Merge Joins

You can enable and disable merge joins at the server and session level using **set sort_merge**, or at the server level with the configuration parameter **enable sort-merge joins and JTC.** This configuration parameter also enables and disables join transitive closure.

Enabling at the Server Level

To enable merge joins server-wide, set **enable sort-merge joins and JTC** to 1. The default value is 0, which means that merge joins are not considered. When this value is set to 1, merge joins and join transitive closure are considered for equijoins. If merge joins are disabled at the server level, they can be enabled for a session with **set sort_merge.**

Join transitive closure can be enabled independently at the session level with **set jtc on**. See “Enabling and Disabling Join Transitive Closure” on page 20-13.

The configuration parameter is dynamic, and can be reset without restarting the server.

Enabling Merge Joins for a Session

To enable merge joins for a session, use:

```
set sort_merge on
```

To disable merge joins during a session, use:

```
set sort_merge off
```

The session setting has precedence over the server-wide setting; you can use merge joins in a session or stored procedure even if they are disabled at the server-wide level.

The Reformatting Strategy

When a table is large and has no useful index for a join, the optimizer considers a sort merge join, and also considers creating and sorting a worktable, and using a nested-loop join.

The process of generating a worktable with a clustered index and performing a nested-loop join is known as **reformatting**.

Like a sort-merge join, reformatting scans the tables and copies qualifying rows to a worktable. But instead of the sort and merge used for a merge join, Adaptive Server creates a temporary clustered index on the join column for the inner table. In some cases, creating and using the clustered index is cheaper than a sort-merge join.

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 a merge join or repeated table scans.

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 17-37.

Subquery Optimization

➤ ***Note***

This section describes the mode of subquery processing that was introduced in SQL Server release 11.0. If your stored procedures, triggers, or 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_procqmode** in the *Adaptive Server Reference Manual* for more information on determining the processing mode.

Subqueries use the following 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 17-47 for an explanation of the **showplan** messages for subquery processing.

Flattening *in*, *any*, and *exists* Subqueries

Adaptive Server can flatten some quantified predicate subqueries to a join. Quantified predicate subqueries are introduced with **in**, **any**, or **exists**. Each result row in the outer query is returned once, and only once, if the subquery condition evaluates to TRUE.

When Flattening Can Be Done

- For any level of nesting of subqueries, for example:

```
select au_lname, au_fname
from authors
where au_id in
   (select au_id
    from titleauthor
    where title_id in
       (select title_id
        from titles
        where type = "popular_comp") )
```

- For multiple subqueries in the outer query, for example:

```
select title, type
from titles
where title in
   (select title
    from titles, titleauthor, authors
    where titles.title_id = titleauthor.title_id
    and titleauthor.au_id = authors.au_id
    and authors.state = "CA")
and title in
   (select title
    from titles, publishers
    where titles.pub_id = publishers.pub_id
    and publishers.state = "CA")
```

Exceptions to Flattening

A subquery introduced with **in**, **any**, or **exists** cannot be flattened if one of the following is true:

- The subquery is correlated and contains one or more aggregates.
- The subquery is in the select list or in the **set** clause of an **update** statement.
- The subquery is connected to the outer query with **or**.
- The subquery is part of an **isnull** predicate.
- The subquery is the outermost subquery in a **case** expression.

If the subquery computes a scalar aggregate, materialization rather than flattening is used. See "Materializing Subquery Results" on page 7-28.

Flattening Methods

Adaptive Server uses one of these flattening methods to resolve a quantified predicate subquery using a join:

- A regular join – If the uniqueness conditions in the subquery mean that it returns a unique set of values, the subquery can be flattened to use a regular join.
- An existence join, also known as a semi-join – Instead of scanning a table to return 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 unique reformat – The subquery result set is selected into a worktable, sorted to remove duplicates, and a clustered index is built on the worktable. The clustered index is used to perform a a regular join.
- A duplicate elimination sort optimization– The subquery is flattened into a regular join that selects the results into a worktable, then the worktable is sorted to remove duplicate rows

Join Order and Flattening Methods

A major factor in the choice of flattening method depends on the cost of the possible join orders. For example, in a join of *t1*, *t2*, and *t3*:

```
select * from t1, t2
where t1.c1 = t2.c1
and t2.c2 in (select c3 from t3)
```

If the cheapest join order is *t1*, *t2*, *t3* or *t2*, *t1*, *t3*, a regular join an existence join is used. However, if it is cheaper to perform the join with *t3* as the outer table, say, *t3*, *t1*, *t2*, a unique reformat or duplicate elimination sort is used.

The resulting flattened join can include nested-loop joins or merge joins. When an existence join is used, merge joins can only be performed before, not after, the existence join.

Flattened Subqueries Executed as Regular Joins

Quantified predicate subqueries can be executed as normal joins when the result set of the subquery is a set of unique values. For example, if there is a unique index on *publishers.pub_id*, this single-table subquery is guaranteed to return a set of unique values:

```
select title
from titles
where pub_id in (select pub_id
    from publishers
    where state = "TX")
```

With a non-unique index on *publishers.city*, this query can also be executed using a regular join:

```
select au_lname
from authors a
where exists (select city
        from publishers p where p.city = a.city)
```

Although the index on *publishers.city* is not unique, the join can still be flattened to a normal join if the index is used to filter duplicate rows from the query.

Diagnostic Messages For Regular Joins

When a subquery is flattened to a normal join, **showplan** output shows a normal join. If filtering is used, **showplan** output is not different; the only diagnostic message is in **dbcc traceon(310)** output, where the *method* for the table indicates “NESTED ITERATION with Tuple Filtering.”

Flattened Subqueries Executed as Existence Joins

All **in**, **any**, and **exists** queries test for the existence of qualifying values and return TRUE as soon as a matching row is found.

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 existence 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.

showplan Messages for Existence Joins

When subqueries are flattened to use existence joins, the **showplan** output shows output for a join, with the message "EXISTS TABLE: nested iteration" as the join type for the table in the subquery.

Flattened Subqueries Executed Using Unique Reformatting

To perform unique reformatting, Adaptive Server:

- Selects rows into a worktable and sorts the worktable, removing duplicates and creating a clustered index on the join key.
- Joins the worktable with the next table in the join order. If there is a nonunique index on *publishers.pub_id*, this query can use a unique reformat strategy:

  ```
  select title_id
  from titles
  where pub_id in
  (select pub_id from publishers where state = "TX")
  ```

  This query is executed as:

  ```
  select pub_id
  into #publishers
  from publishers
  where state = "TX"
  ```

And after the sort removes duplicates and creates the clustered index:

```
select title_id
from titles, #publishers
where titles.pub_id = #publishers.pub_id
```

Diagnostic Messages for Unique Reformatting

showplan messages for unique reformatting show “Worktable created for REFORMATTING” in Step 1, and “Using Clustered Index” on the worktable in Step 2.

dbcc traceon(310) displays “REFORMATTING with Unique Reformatting” for the method for the *publishers* table.

Flattened Subqueries Using Duplicate Elimination d

When it is cheaper to place the subquery tables as outer tables in the join order, the query is executed by:

- Performing a regular join with the subquery flattened into the outer query, placing results in a worktable.
- Sorting the worktable to remove duplicates.

For example, *salesdetail* has duplicate values for *title_id*, and it is used in this subquery:

```
select  title_id, au_id, au_ord
from titleauthor ta
where title_id in (select ta.title_id
    from titles t, salesdetail sd
    where t.title_id = sd.title_id
    and ta.title_id = t.title_id
    and type = 'travel' and qty > 10)
```

If the best join order for this query is *salesdetail*, *titles*, *titleauthor*, the optimal join order can be used by:

- Selecting all of the query results into a worktable
- Removing the duplicates from the worktable and returning the results to the user

showplan Messages for Flattened Subqueries Performing Sorts

showplan output includes two steps for subqueries that use normal joins plus a sort. The first step shows “Worktable1 created for DISTINCT” and the flattened join. The second step shows the sort and select from the worktable.

dbcc traceon(310) prints a message for each join permutation when a table or tables from a quantified predicate subquery is placed first in the join order. Here is the output when the join order used for the query above is considered:

```
2 - 0 - 1 -

This join order created while converting an exists
join to a regular join, which can happen for
subqueries, referential integrity, and select
distinct.
```

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, a subquery is processed in two steps: the results from the inner query are **materialized**, or stored in a temporary worktable or internal variable, 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 query is executed by:

- Executing the subquery and storing the result in an internal variable.

- Substituting 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. The total cost of a materialized expression subquery is the sum of the cost of the two separate queries.

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
```

The total cost of executing quantified predicate subqueries is the sum of the query costs for the two steps.

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 less expensive clauses can 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 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. This provides a performance improvement, because expensive subqueries need to be executed less often.

Subquery Introduced with an *and* Clause

When **and** joins the clauses, evaluation stops as soon as any clause evaluates to FALSE. The row is skipped.

This query contains two **and** clauses, in addition to the correlated 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, after it evaluates the conditions on *price* and *au_ord*. 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, and the row is returned.

This query contains two **or** 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)
```

Adaptive Server orders the conditions in the query plan to evaluate the subquery last. If a row meets the condition of the **or** clause, Adaptive Server returns the row without executing 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. 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 staticstics 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.

Optimizing Subqueries

When queries containing subqueries are not flattened or materialized:

- The outer query and each unflattened subquery are 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 “nested over” 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 nested over.

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 optimizes search arguments that are linked with **or**. This description applies only to join clauses.

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** 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.

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.

8 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?

A cursor is a symbolic name that is associated with a **select** statement. It enables you to access the results of a **select** statement one row at a time. Figure 8-1 shows a cursor accessing the *authors* table.

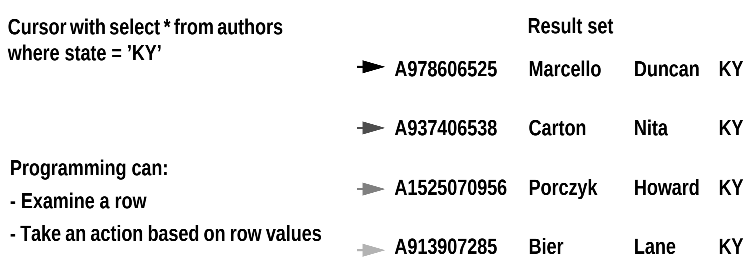


Figure 8-1: Cursor example

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 8-2 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.

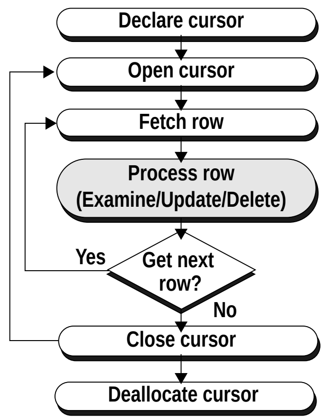


Figure 8-2: Cursor flowchart

Cursors: A Simple Example

Here is a simple example of a cursor with the “Process Rows” step from Table 8-2 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 open a cursor, memory is allocated to the cursor and to store the query plan that is generated. While the cursor is open, Adaptive Server holds intent table locks and sometimes row or page locks. Table 8-3 shows the duration of locks during cursor operations.

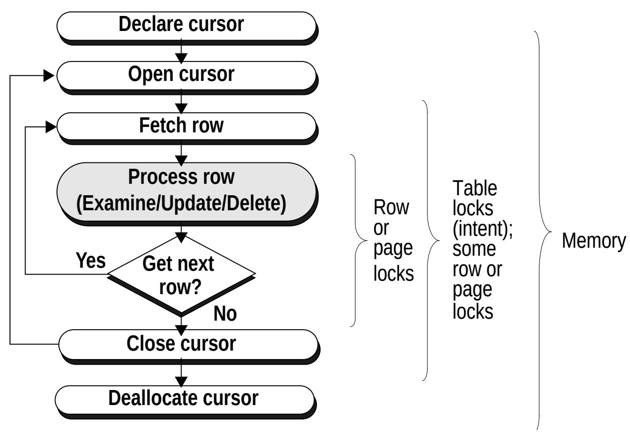


Figure 8-3: Resource use by cursor statement

The memory resource descriptions in Figure 8-3 and Table 8-1 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 8-5.

Table 8-1: Locks and memory use for isql and Client-Library client cursors

Cursor Command	Resource Use
declare cursor	When you declare a cursor, Adaptive Server uses only enough memory to store the query text.
open	When you open a cursor, Adaptive Server allocates memory to the cursor and to store the query plan that is generated. The server optimizes the query, traverses indexes, and sets up memory variables. The server does not access rows yet, unless it needs to build worktables. However, it does set up the required table-level locks (intent locks). Row and page locking behavior depends on the isolation level, server configuration, and query type. See “Lock Types and Duration During Query Processing” on page 26-23 for more information.
fetch	When you execute a **fetch**, Adaptive Server gets the row(s) required and reads specified values into the cursor variables or sends the row to the client. If the cursor needs to hold lock on rows or pages, the lock are held until a **fetch** moves the cursor off the row or page or until the cursor is closed. The lock is either a shared or an update lock, depending on how the cursor is written.
close	When you close a cursor, Adaptive Server releases the locks and some of the memory allocation. You can open the cursor again, if necessary.
deallocate cursor	When you deallocate a cursor, Adaptive Server releases the rest of the memory resources used by the cursor. To reuse the cursor, you must declare it again.

Memory Use and Execute Cursors

The descriptions of **declare cursor** and **deallocate cursor in** Table 8-1 refer to ad hoc cursors that are sent by **isql** or Client-Library. Other kinds of cursors allocate memory differently:

- For cursors that are declared **on** stored procedures, only a small amount of memory is allocated at **declare cursor** time. Cursors declared on stored procedures are sent using Client-Library or the precompiler and are known as **execute cursors**.
- For cursors declared **within** a stored procedure, memory is already available for the stored procedure, and the **declare** statement does not require additional memory.

Cursor Modes: 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 or row locks. If **read committed with lock** is set to 0, and the query runs at isolation level 1, it uses instant duration locks, and does not hold the page or row locks until the next fetch.

Read-only mode is in effect when you specify **for read only** or when the cursor's select statement uses **distinct**, **group by**, **union**, or aggregate functions, and in some cases, an **order by** clause.

Update mode uses update page or row locks. It is in effect when:

- You specify **for update**.
- The **select** statement does not include **distinct**, **group by**, **union**, a subquery, aggregate functions, or the **at isolation read uncommitted** clause.
- You specify **shared**.

If *column_name_list* is specified, only those columns are updatable.

For more information on locking during cursor processing, see "Lock Types and Duration During Query Processing" on page 26-23.

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

When a query is used in a cursor, it may require or choose different indexes than the same query used outside of a cursor.

Allpages-Locked Tables

For read-only cursors, queries at isolation level 0 (dirty reads) require a unique index. Read-only cursors at isolation level 1 or 3 should produce the same query plan as the **select** statement outside of a cursor.

The index requirements for updatable cursors mean that updatable cursors may use different query plans than read-only cursors. Update cursors have these indexing requirements:

- If the cursor is not declared for update, a unique index is preferred over a table scan or a nonunique index. 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 on allpages-locked tables. An error is raised if no unique index exists.
- If the cursor is declared for update with a **for update of** list, then only a unique index **without** any columns from the list can be chosen on an allpages-locked table. An error is raised if no unique index qualifies.

When cursors are involved, an index that contains an IDENTITY column is considered unique, even if the index is not declared unique. In some cases, IDENTITY columns must be added to indexes to make them unique, or the optimizer might be forced to choose a suboptimal query plan for a cursor query.

Data-Only-Locked Tables

In data-only-locked tables, fixed row IDs are used to position cursor scans, so unique indexes are not required for dirty reads or updatable cursors. The only cause for different query plans in updatable cursors is that table scans are used if columns from only useful index are included in the **for update of** list.

Table Scans to Avoid the Halloween Problem

The Halloween problem is an update anomaly that can occur when a client using a cursor updates a column of the cursor result set row, and that column defines the order in which the rows are returned from the table. For example, if a cursor was to use an index on *last_name*, *first_name*, and update one of these columns, the row could appear in the result set a second time.

To avoid the Halloween problem on data-only-locked tables, Adaptive Server chooses a table scan when the columns from an otherwise useful index are included in the column list of a **for update** clause.

For implicitly updatable cursors declared without a **for update** clause, and for cursors where the column list in the **for update** clause is empty, cursors that update a column in the index used by the cursor may encounter the Halloween problem.

Comparing Performance with and Without Cursors

This section examines the performance of a stored procedure written two different ways:

- Without a cursor – This procedure scans the table three times, changing the price of each book.
- With a cursor – This procedure makes only one pass through the table.

In both examples, there is a unique index on *titles(title_id)*.

Sample Stored Procedure Without a Cursor

This is an example of a stored procedure without cursors:

```
/* Increase the prices of books in the
** titles table as follows:
**
** If current price is <= $30, increase it by 20%
** If current price is > $30 and <= $60, increase
** it by 10%
** If current price is > $60, increase it by 5%
**
** All price changes must take effect, so this is
** done in a single transaction.
*/

create procedure increase_price
as

    /* start the transaction */
    begin transaction
    /* first update prices > $60 */
    update titles
        set price = price * 1.05
        where price > $60

    /* next, prices between $30 and $60 */
    update titles
```

```
        set price = price * 1.10
    where price > $30 and price <= $60

    /* and finally prices <= $30 */
    update titles
    set price = price * 1.20
    where price <= $30

    /* commit the transaction */
    commit transaction

return
```

Sample Stored Procedure With a Cursor

This procedure performs the same changes to the underlying table as the procedure written without a cursor, but it uses cursors instead of set-oriented programming. As each row is fetched, examined, and updated, a lock is held on the appropriate data page. Also, as the comments indicate, each update commits as it is made, since there is no explicit transaction.

```
/* Same as previous example, this time using a
** cursor. Each update commits as it is made.
*/
create procedure increase_price_cursor
as
declare @price money

/* declare a cursor for the select from titles */
declare curs cursor for
    select price
    from titles
    for update of price

/* open the cursor */
open curs

/* fetch the first row */
fetch curs into @price

/* now loop, processing all the rows
** @@sqlstatus = 0 means successful fetch
** @@sqlstatus = 1 means error on previous fetch
** @@sqlstatus = 2 means end of result set reached
*/
while (@@sqlstatus != 2)
```

```
begin
    /* check for errors */
    if (@@sqlstatus = 1)
    begin
        print "Error in increase_price"
        return
    end

    /* next adjust the price according to the
    ** criteria
    */
    if @price > $60
    select @price = @price * 1.05
    else
    if @price > $30 and @price <= $60
    select @price = @price * 1.10
    else
    if @price <= $30
    select @price = @price * 1.20

    /* 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 8-2 shows statistics gathered against a 5000-row table. Note that the cursor code takes over 4 times longer, even though it scans the table only once.

Table 8-2: Sample execution times against a 5000-row table

Procedure	Access Method	Time
increase_price	Uses three table scans	28 seconds
increase_price_cursor	Uses cursor, single table scan	125 seconds

Results from tests like these can vary widely. They are most pronounced on systems that have busy networks, a large number of active database users, and multiple users accessing the same table.

Cursor vs. Non-Cursor Performance Explanation

In addition to locking, cursors involve more network activity than set operations and incur the overhead of processing instructions. The application program needs to communicate with Adaptive Server regarding every result row of the query. This is why the cursor code took much longer to complete than the code that scanned the table three times.

When cursors are absolutely necessary, 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

If there is a set level programming equivalent, it may be preferable, even if it involves multiple table scans.

Locking with Read-Only Cursors

Here is a piece of cursor code you can use to display the locks that are set up at each point in the life of a cursor. The following example uses an allpages-locked table. Execute the code in Figure 8-4, 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 8-4: Read-only cursors and locking experiment input

Table 8-3 shows the results.

Table 8-3: Locks held on data and index pages by cursors

Event	Data Page
After **declare**	No cursor-related locks.
After **open**	Shared intent lock on *authors*.
After first **fetch**	Shared intent lock on *authors* and shared page lock on a page in *authors*.
After 100 fetches	Shared intent lock on *authors* and shared page lock on a different page in *authors*.
After **close**	No cursor-related locks.

If you issue another **fetch** command after the last row of the result set has been fetched, the locks on the last page are released, so there will be no cursor-related locks.

With a data-only-locked table:

- If the cursor query runs at isolation level 1, and **read committed with lock** is set to 0, you do not see any page or row locks. The values are copied from the page or row, and the lock is immediately released.

- If **read committed with lock** is set to 1 or if the query runs at isolation level 2 or 3, you see either shared page or shared row locks at the point that Table 8-3 indicates shared page locks. If the table uses datarows locking, the **sp_lock** report includes the row ID of the fetched row.

Isolation Levels and Cursors

The query plan for a cursor is compiled and optimized at the time the cursor is opened. You cannot open a cursor and then use **set transaction isolation level** to change the isolation level at which the cursor operates.

Since cursors using isolation level 0 are compiled differently from those using other isolation levels, you cannot open a cursor at isolation level 0 and open or fetch from it at level 1 or 3. Similarly, you cannot open a cursor at level 1 or 3 and then fetch from it at level 0. Attempts to fetch from a cursor at an incompatible level result in an error message.

Once the cursor has been opened at a particular isolation level, you must deallocate the cursor before changing isolation levels. The effects of changing isolation levels while the cursor is open are as follows:

- Attempting to close and reopen the cursor at another isolation level fails with an error message.
- Attempting to change isolation levels without closing and reopening the cursor has no effect on the isolation level in use and does not produce an error message.

You can include an **at isolation** clause in the cursor to specify an isolation level. The cursor in the example below can be declared at level 1 and fetched from level 0 because the query plan is compatible with the isolation level:

```
declare cprice cursor for
select title_id, price
    from titles
    where type = "business"
    at isolation read uncommitted
```

Partitioned Heap Tables and Cursors

A cursor scan of an unpartitioned heap table can read all data up to and including the final insertion made to that table, even if insertions took place after the cursor scan started.

If a heap table is partitioned, data can be inserted into one of the many page chains. The physical insertion point may be before or after the current position of a cursor scan. This means that a cursor scan against a partitioned table is **not** guaranteed to scan the final insertions made to that table.

➤ ***Note***

If your cursor operations require all inserts to be made at the end of a single page chain, **do not** partition the table used in the cursor scan.

Optimizing Tips for Cursors

Here are several optimizing tips for cursors:

- Optimize cursor selects using the cursor, not an ad hoc query.
- Use **union** or **union all** instead of **or** clauses or **in** lists.
- Declare the cursor's intent.
- Specify column names in the **for update** clause.
- Fetch more than one row if you are returning rows to the client.
- Keep cursors open across commits and rollbacks.
- Open multiple cursors on a single connection.

Optimizing for Cursor Selects Using a Cursor

A standalone **select** statement may be optimized very differently than the same **select** statement in an implicitly or explicitly updatable cursor. When you are developing applications that use cursors, always check your query plans and I/O statistics using the cursor, rather than using a standalone **select**. In particular, index restrictions of updatable cursors require very different access methods.

Using *union* Instead of *or* Clauses or *in* Lists

Cursors cannot use the dynamic index of row IDs generated by the OR strategy. Queries that use the OR strategy in standalone **select** statements usually perform table scans using read-only cursors. Updatable cursors may need to use a unique index and still require access to each data row, in sequence, in order to evaluate the query clauses. See “Access Methods and Costing for or and in Clauses” on page 6-23 for more information.

A read-only cursor using **union** creates a worktable when the cursor is declared, and sorts it to remove duplicates. Fetches are performed on the worktable. A cursor using **union all** can return duplicates and does not require a worktable.

Declaring the Cursor’s Intent

Always declare a cursor’s intent: read-only or updatable. This gives you greater control over concurrency implications. If you do not specify the intent, Adaptive Server decides for you, and very often it chooses updatable cursors. Updatable cursors use update locks, thereby preventing other update locks or exclusive locks. If the update changes an indexed column, the optimizer may need to choose a table scan for the query, resulting in potentially difficult concurrency problems. Be sure to examine the query plans for queries that use updatable cursors.

Specifying Column Names in the *for update* Clause

Adaptive Server acquires update locks on the pages or rows of all tables that have columns listed in the **for update** clause of the cursor **select** statement. If the **for update** clause is not included in the cursor declaration, all tables referenced in the **from** clause acquire update locks.

The following query includes the name of the column in the **for update** clause, but acquires update locks only on the *titles* table, since *price* is mentioned in the **for update** clause. The table use allpages locking. The locks on *authors* and *titleauthor* are shared page locks:

```
declare curs3 cursor
for
select au_lname, au_fname, price
    from titles t, authors a,
        titleauthor ta
where advance <= $1000
    and t.title_id = ta.title_id
    and a.au_id = ta.au_id
for update of price
```

Table 8-4 shows the effects of:

- Omitting the **for update** clause entirely—no **shared** clause

- Omitting the column name from the **for update** clause
- Including the name of the column to be updated in the **for update** clause
- Adding **shared** after the name of the *titles* table while using **for update of price**

In this table, the additional locks, or more restrictive locks for the two versions of the **for update** clause are emphasized.

Table 8-4: Effects of for update clause and shared on cursor locking

Clause	*titles*	*authors*	*titleauthor*
None		sh_page on index	
	sh_page on data	sh_page on data	sh_page on data
for update	**updpage on index**	**updpage on index**	
	updpage on data	**updpage on data**	**updpage on data**
for update of price		sh_page on index	
	updpage on data	sh_page on data	sh_page on data
for update of price + **shared**		sh_page on index	
	sh_page on data	sh_page on data	sh_page on data

Using *set cursor rows*

The SQL standard specifies a one-row fetch for cursors, which wastes network bandwidth. Using the **set cursor rows** query option and Open Client's transparent buffering of fetches, you can improve performance:

```
ct_cursor(CT_CURSOR_ROWS)
```

Be careful when you choose the number of rows returned for frequently executed applications using cursors—tune them to the network. See "Changing Network Packet Sizes" on page 36-3 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 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.

9 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

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 data page access completely, when an index covers a query

- Use ordered data to avoid sorts or to favor merge joins over nested-loop joins

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

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.

If a query is searching for a unique value in a table that has 600 data pages, this requires 600 physical and logical 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.

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.

showplan output reports whether the table is being accessed via a table scan or index. If you think that an index should be used, but **showplan** reports a table scan, **dbcc traceon(302)** output can help you determine the reason. It displays the costing computations for all optimizable query clauses. If a clause is not included in **dbcc traceon(302)** output, there may be problems with the way the clause is written. If a clause that you think should limit the scan is included in **dbcc traceon(302)** output, look carefully at its costing, and that of the chosen plan reported with **dbcc traceon(310)**.

Index Is Not Selective Enough

An index is selective if it helps the optimizer find a particular row or a set of rows. An index on a unique identifier such as a social security number is highly selective, since it lets the optimizer pinpoint a single row. An index on a nonunique entry such as sex (M, F) is not very selective, and the optimizer would use such an index only in very special cases.

Index Does Not Support Range Queries

Generally, clustered indexes and covering indexes provide good performance for range queries and for SARGs that match many rows. Range queries that reference the keys of noncovering indexes use the index for ranges that return a limited number of rows. As the number of rows the query returns increases, however, using a nonclustered index or a clustered index on a data-only-locked table can cost more than a table scan.

Too Many Indexes Slow Data Modification

If data modification performance is poor, you may have too many indexes.While indexes favor select operations, they slow down data modifications. Every insert or delete operation affects the leaf level, (and sometimes higher levels) of a clustered index on a data-only-locked table, and each nonclustered index, for any locking scheme. Updates to clustered index keys on allpages-locked tables can move the rows to different pages, requiring an update of every nonclustered index. Analyze the requirements for each index and try to eliminate those that are unnecessary or rarely used.

Index Entries Are Too Large

Try to keep index entries as small as possible. You can create indexes with keys up to 600 bytes, but those indexes can store very few rows per index page, which increases the amount of disk I/O needed during queries. The index has more levels, and each level has more pages. Nonmatching index scans can be very expensive.

The following example uses values reported by **sp_estspace** to demonstrate how the number of index pages and leaf levels required increases with key size. It creates nonclustered indexes using 10-, 20, and 40-character keys.

```
create table demotable (c10 char(10),
                        c20 char(20),
                        c40 char(40))

create index t10 on demotable(c10)

create index t20 on demotable(c20)

create index t40 on demotable(c40)

sp_estspace demotable, 500000
```

Table 9-1 shows the results.

Table 9-1: Effects of key size on index size and levels

Index, key size	Leaf-level pages	Index levels
t10, 10 bytes	4311	3
t20, 20 bytes	6946	3
t40, 40 bytes	12501	4

The output shows that the indexes for the 10-column and 20-column keys each have three levels, while the 40-column key requires a fourth level.

The number of pages required is more than 50 percent higher at each level. A nonmatching index scan on the leaf level of *t20* would require 6946 disk reads, compared to 4311 for the index on *t10*.

Exception for Wide Data Rows and Wide Index Rows

Indexes with wide rows may be useful for a small subset of tables and applications:

- The table has very wide rows, resulting in very few rows per data page.
- The set of queries run on the table provides logical choices for a covering index.
- Queries return a sufficiently large number of rows.

For example, if a table has very long rows, and only one row per page, a query that needs to return 100 rows needs to access 100 data pages. An index that covers this query, even with long index rows, could improve performance. For example, if the index rows were 240 bytes, the index would store 8 rows per page, and the query would need to access only 12 index pages.

Index Limits and Requirements

The following limits apply to indexes in Adaptive Server:

- You can create only one clustered index per table, since the data for a clustered index is ordered by index key.
- You can create a maximum of 249 nonclustered indexes per table.
- A key can be made up of as many as 31 columns. The maximum number of bytes per index key is 600.
- When you create a clustered index, Adaptive Server requires empty free space to copy the rows in the table, allocate space for the clustered index pages, and space to re-create any nonclustered indexes on the table. Space required can vary, depending on how full the table's pages are when you begin and what space management properties are applied to the table and index pages. 120% of the current space usage is a general rule; if you have very long keys, you may need even more free space; if many pages are half-empty, the copy takes less room than the

current space. See “Determining the Space Available for Maintenance Activities” on page 30-10 for more information.

- The referential integrity constraints **unique** and **primary key** create unique indexes to enforce their restrictions on the keys. By default, **unique** constraints create nonclustered indexes and **primary key** constraints create clustered indexes.

Choosing Indexes

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 ratio of select operations to data modifications performed on the table?
- Has a clustered index been created for the table?
- Can the clustered index be replaced by a nonclustered index?
- Do any of the indexes cover one or more of the critical queries?
- Is a composite index required to enforce the uniqueness of a compound primary key?
- What indexes can be defined as unique?
- What are the major sorting requirements?
- Do some queries using descending ordering of result sets?
- Do the indexes support 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.

Choose index keys for performance reasons. Create indexes on columns that support the joins, search arguments, and ordering requirements in queries. A common error is to create the clustered index for a table on the primary key, even though it is never used for range queries or ordering result sets.

Guidelines for Clustered Indexes

These are general guidelines for clustered indexes:

- Most allpages-locked tables should have clustered indexes or use partitions to reduce contention on the last page of heaps. In a high-transaction environment, the locking on the last page severely limits throughput.
- If your environment requires a lot of inserts, do not place the clustered index key on a 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. This problem is less severe on data-only-locked tables, but is a major source of lock contention on allpages-locked tables.
- Clustered indexes provide very good performance when the key matches the search argument in range queries, such as:

  ```
  where colvalue >= 5 and colvalue < 10
  ```

  In allpages-locked tables, rows are maintained in key order and pages are linked in order, providing very fast performance for queries using a clustered index. In data-only-locked tables, rows are in key order after the index is created, but the clustering can decline over time.

- Other good choices for clustered index keys are columns used in **order by** clauses and in joins.
- If possible, do not include frequently updated columns as keys in clustered indexes on allpages-locked tables. When the keys are updated, the rows must be moved from the current location to a new page. Also, if the index is clustered, but not unique, updates are done in deferred mode.

Choosing Clustered Indexes

Choose indexes based on the kinds of **where** clauses or joins you perform. Choices for clustered indexes are:

- The primary key, if it is used for **where** clauses and if it randomizes inserts
- Columns that are accessed by range, such as:

```
col1 between 100 and 200
col12 > 62 and < 70
```

- Columns used by **order by**
- Columns that are not frequently changed
- Columns used in joins

If there are several possible choices, choose the most commonly needed physical order as a first choice. As a second choice, look for range queries. During performance testing, check for "hot spots" due to lock contention.

Candidates for Nonclustered Indexes

When choosing columns for nonclustered indexes, consider all the uses that were not satisfied by your clustered index choice. In addition, look at columns that can provide performance gains through index covering.

On data-only-locked tables, clustered indexes can perform index covering, since they have a leaf level above the data level. On allpages-locked tables, noncovered range queries work well for clustered indexes, but may or may not be supported by nonclustered indexes, depending on the size of the range.

Consider 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, define it as unique so the optimizer knows immediately that only one row matches a search argument or a join on the key.
- If your database design uses referential integrity (the **references** keyword or the **foreign key...references** keywords in the **create table** statement), the referenced columns **must** have a unique index, or the attempt to create the referential integrity constraint fails. However, Adaptive Server does not automatically create an index on the referencing column. If your application updates primary keys or deletes rows from primary key tables, you may want to create an index on the referencing column so that these lookups do not perform a table scan.
- If your applications use cursors, see “Index Use and Requirements for Cursors” on page 8-6.
- 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.
- Use small datatypes whenever it fits your design.
  - Numerics compare slightly faster than strings internally.
  - Variable-length character and binary types require more row overhead than fixed-length types, so if there is little difference between the average length of a column and the defined length, use fixed length. Character and binary types that accept null values are variable-length by definition.

- Whenever possible, use fixed-length, non-null types for short columns that will be used as index keys.

- Be sure that the datatypes of the join columns in different tables are compatible. If Adaptive Server has to convert a datatype on one side of a join, it may not use an index for that table. See "Datatype Mismatches and Query Optimization" on page 5-21 for more information.

Choosing Nonclustered Indexes

When you consider adding nonclustered indexes, you must weigh the improvement in retrieval time against the increase in data modification time. In addition, you need to consider these questions:

- How much space will the indexes use?
- How volatile is the candidate column?
- How selective are the index keys? Would a scan be better?
- Are there a lot of duplicate values?

Because of data modification overhead, add nonclustered indexes only when your testing shows that they are helpful.

Performance Price for Data Modification

Each nonclustered index needs to be updated, for all locking schemes:

- For each insert into the table
- For each delete from the table

An update to the table that changes part of an index's key requires updating just that index.

For tables that use allpages locking, all indexes need to be updated:

- For any update that changes the location of a row by updating a clustered index key so that the row moves to another page
- For every row affected by a data page split

For allpages-locked tables, exclusive locks are held on affected index pages for the duration of the transaction, increasing lock contention as well as processing overhead.

Some applications experience unacceptable performance impacts with only three or four indexes on tables that experience heavy data modification. Other applications can perform well with many more tables.

Choosing Composite Indexes

If your analysis shows that more than one column is a good candidate for a clustered index key, you may be able to provide clustered-like access with a composite index that covers a particular query or set of queries. These include:

- Range queries.
- Vector (grouped) aggregates, if both the grouped and grouping columns are included. Any search arguments must also be included in the index.
- Queries that return a high number of duplicates.
- Queries that include **order by**.
- Queries that table scan, but use a small subset of the columns on the table.

Tables that are read-only or read-mostly can be heavily indexed, as long as your database has enough space available. If there is little update activity and high select activity, you should provide indexes for all of your frequent queries. Be sure to test the performance benefits of index covering.

Key Order and Performance in Composite Indexes

Covered queries can provide excellent response time for specific queries when the leading columns are used.

With the composite nonclustered index on *au_lname, au_fname, au_id*, this query runs very quickly:

```
select au_id
    from authors
where au_fname = "Eliot" and au_lname = "Wilk"
```

This covered point query needs to read only the upper levels of the index and a single page in the leaf-level row in the nonclustered index of a 5000-row table.

This similar-looking query (using the same index) does not perform quite as well. This query is still covered, but searches on *au_id*:

```
select au_fname, au_lname
    from authors
where au_id = "A1714224678"
```

Since this query does not include the leading column of the index, it has to scan the entire leaf level of the index, about 95 reads.

Adding a column to the select list in the query above, which may seem like a minor change, makes the performance even worse:

```
select au_fname, au_lname, phone
    from authors
where au_id = "A1714224678"
```

This query performs a table scan, reading 222 pages. In this case, the performance is noticeably worse. For any search argument that is not the leading column, Adaptive Server has only two possible access methods: a table scan, or a covered index scan. It does not scan the leaf level of the index for a non-leading search argument and then access the data pages. A composite index can be used only when it covers the query or when the first column appears in the **where** clause.

For a query that includes the leading column of the composite index, adding a column that is not included in the index adds only a single data page read. This query must read the data page to find the phone number:

```
select au_id, phone
    from authors
where au_fname = "Eliot" and au_lname = "Wilk"
```

Table 9-2 shows the performance characteristics of different **where** clauses with a nonclustered index on *au_lname, au_fname, au_id* and no other indexes on the table.

Table 9-2: Composite nonclustered index ordering and performance

Columns in the *where* Clause	Performance with the Indexed Columns in the Select List	Performance with Other Columns in the Select List
au_lname or *au_lname, au_fname* or *au_lname, au_fname, au_id*	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)
au_fname or *au_id* or *au_fname, au_id*	Moderate; index is scanned to return values	Poor; index not used, table scan

Choose the ordering of the composite index so that most queries form a prefix subset.

Advantages of Composite Indexes

Composite indexes have these advantages:

- A composite index provides opportunities for index covering.
- If queries provide search arguments on each of the keys, the composite index requires fewer I/Os than the same query using an index on any single attribute.
- A composite index is a good way to enforce the uniqueness of multiple attributes.

Good choices for composite indexes are:

- Lookup tables
- Columns that are frequently accessed together
- Columns used for vector aggregates
- Columns that make a frequently used subset from a table with very wide rows

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 nearly as wide as the table
- Composite indexes where only a minor key is used in the **where** clause

Techniques for Choosing Indexes

This section presents a study of two queries that must access a single table, and the indexing choices for these two queries. The two queries are:

- A range query that returns a large number of rows
- A point query that returns only one or two rows

Choosing an Index for a Range Query

Assume that you need to improve the performance of the following query:

```
select title
from titles
where price between $20.00 and $30.00
```

Some basic statistics on the table are:

- The table has 1,000,000 rows, and uses allpages locking.
- There are 10 rows per page; pages are 75 percent full, so the table has approximately 135,000 pages.
- 190,000 (19%) of the titles are priced between $20 and $30.

With no index, the query would scan all 135,000 pages.

With a clustered index on *price*, the query would find the first $20 book and begin reading sequentially until it gets to the last $30 book. With pages about 75 percent full, the average number of rows per page is 7.5. To read 190,000 matching rows, the query would read approximately 25,300 pages, plus 3 or 4 index pages.

With a nonclustered index on *price* and random distribution of *price* values, using the index to find the rows for this query would requires reading about 19 percent of the leaf level of the index, about 1,500 pages. If the price values are randomly distributed, the number of data pages that must be read is likely to be high, perhaps as many data pages as there are qualifying rows, 190,000. Since a table scan requires only 135,000 pages, 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 covering nonclustered index on *price, title* is best.

Adding a Point Query with Different Indexing Requirements

The index choice for the range query on *price* produced a clear performance choice when all possibly useful indexes were considered. Now, assume this query also needs to run against *titles*:

```
select price
from titles
where title = "Looking at Leeks"
```

You know that there are very few duplicate titles, so this query returns only one or two rows.

Considering both this query and the previous query, Table 9-3 shows four possible indexing strategies and estimate costs of using each index. The estimates for the numbers of index and data pages were generated using a fillfactor of 75 percent with **sp_estspace**:

```
sp_estspace titles, 1000000, 75
```

The values were rounded for easier comparison.

Table 9-3: Comparing index strategies for two queries

Possible Index Choice		Index Pages	Range Query on *price*	Point Query on *title*
1	Nonclustered on *title* Clustered on *price*	36,800 650	Clustered index, about 26,600 pages (135,000 * .19) With 16K I/O: 3,125 I/Os	Nonclustered index, 6 I/Os
2	Clustered on *title* Nonclustered on *price*	3,770 6,076	Table scan, 135,000 pages With 16K I/O: 17,500 I/Os	Clustered index, 6 I/Os
3	Nonclustered on *title, price*	36,835	Nonmatching index scan, about 35,700 pages With 16K I/O: 4,500 I/Os	Nonclustered index, 5 I/Os
4	Nonclustered on *price, title*	36,835	Matching index scan, about 6,800 pages (35,700 * .19) With 16K I/O: 850 I/Os	Nonmatching index scan, about 35,700 pages With 16K I/O: 4,500 I/Os

Examining the figures in Figure 9-3 shows that:

- For the range query on *price*, choice 4 is best; choices 1 and 3 are acceptable with 16K I/O.
- For the point query on *titles*, indexing choices 1, 2, and 3 are excellent.

The best indexing strategy for a combination of these two queries is to use two indexes:

- Choice 4, for range queries on *price*.
- Choice 2, for point queries on *title*, since the clustered index requires very little space.

You may need additional information to help you determine which indexing strategy to use to support multiple queries. Typical considerations are:

- What is the frequency of each query? How many times per day or per hour is the query run?
- What are the response time requirements? Is one of them especially time critical?
- What are the response time requirements for updates? Does creating more than one index slow updates?
- Is the range of values typical? Is a wider or narrower range of prices, such as $20 to $50, often used? How do different ranges affect index choice?
- Is there a large data cache? Are these queries critical enough to provide a 35,000-page cache for the nonclustered composite indexes in index choice 3 or 4? Binding this index to its own cache would provide very fast performance.
- What other queries and what other search arguments are used? Is this table frequently joined with other tables?

Index and Statistics Maintenance

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.
- Use space management properties to reduce page splits and to reduce the frequency of maintenance operations.

Monitoring Applications and Indexes over Time

Periodically, check the query plans, as described in Chapter 17, “Using set showplan,” and the I/O statistics for your most frequent user queries. Pay special attention to noncovering indexes that support range queries. They are most likely to switch to table scans if the data distribution changes.

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. Indexes should be based on the transactions and processes that are being run, not on the original database design.

Check query plans to determine whether your indexes are being used.

Maintaining Index and Column Statistics

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

- After deleting or inserting rows that change the skew of key values in the index
- After adding rows to a table whose rows were previously deleted with **truncate table**
- After updating values in index columns

Run **update statistics** after inserts to any index that includes an IDENTITY column or any increasing key value. Date columns often have regularly increasing keys. Running **update statistics** on these types of indexes is especially important if the IDENTITY column or other increasing key is the leading column in the index. After a number of rows have been inserted past the last key in the table when the index was created, all that the optimizer can tell is that the search value lies beyond the last row in the distribution page. It cannot accurately determine how many rows match a given value.

➤ ***Note***

Failure to update statistics can severely hurt performance.

See Chapter 10, "Managing Statistics to Improve Performance," for more information.

Rebuilding Indexes

Rebuilding indexes reclaims space in the B-trees. As pages are split and rows are deleted, indexes may contain many pages that 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.

You can rebuild indexes by dropping and re-creating the index. If the table uses data-only locking, you can run the **reorg rebuild** command on the table or on an individual index.

Re-create or rebuild indexes when:

- Data and usage patterns have changed significantly.
- A period of heavy inserts is expected, or has just been completed
- The sort order has changed.
- Queries that use large I/O require more disk reads than expected, or **optdiag** reports lower cluster ratios than usual.
- Space usage exceeds estimates because heavy data modification has left many data and index pages partially full.
- Space for expansion provided by the space management properties (fillfactor, expected row size, and reserve page gap) has been filled by inserts and updates, resulting in page splits, forwarded rows, and fragmentation.
- **dbcc** has identified errors in the index.

If you re-create a clustered index or **run reorg** rebuild on a data-only-locked table, all nonclustered indexes are re-created, since creating the clustered index moves rows to different pages. Nonclustered indexes must be re-created to point to the correct pages.

In many database systems, there are well-defined peak periods and off-hours. You can use off-hours to your advantage; for example:

- To delete all indexes to allow more efficient bulk inserts
- To create a new group of indexes to help generate a set of reports

See "Creating Indexes" on page 30-2 for information about configuration parameters that increase the speed of creating indexes.

Speeding Index Creation with *sorted_data*

If data is already sorted, you can use the **sorted_data** option for the **create index** command to save index creation time. You can use this option for both clustered and nonclustered indexes. See “Creating an Index on Sorted Data” on page 30-3 for more information.

Choosing Space Management Properties for Indexes

Space management properties can help reduce the frequency of index maintenance. In particular, fillfactor can reduce the number of page splits on leaf pages of nonclustered indexes and on the data pages of allpages-locked tables with clustered indexes. See Chapter 31, “Setting Space Management Properties,” for more information on choosing fillfactor values for indexes.

Additional Indexing Tips

Here are some additional suggestions that can lead to improved performance when you are creating and using indexes:

- Modify the logical design to make use of an artificial column and a lookup table for tables that require a large index entry.
- Reduce the size of an index entry for a frequently used index.
- Drop indexes during periods when frequent updates occur and rebuild them before periods when frequent selects occur.
- If you do frequent index maintenance, configure your server to speed up the sorting. See “Configuring Adaptive Server to Speed Sorting” on page 30-2 for information about configuration parameters that enable faster sorting.

Creating Artificial Columns

When indexes become too large, especially composite indexes, it is beneficial to create an artificial column that is assigned to a row, with a secondary lookup table that is used to translate between the internal ID and the original columns. This may increase response time for certain queries, but the overall performance gain due to a more compact index and shorter data rows is usually worth the effort.

Keeping Index Entries Short and Avoiding Overhead

Avoid storing purely numeric IDs as character data. Use integer or numeric IDs whenever possible to:

- Save storage space on the data pages
- Make index entries more compact
- Improve performance, since internal comparisons are faster

Index entries on *varchar* columns require more overhead than entries on *char* columns. For short index keys, especially those with little variation in length in the column data, use *char* for more compact index entries.

Dropping and Rebuilding Indexes

You might drop nonclustered indexes prior to a major set of inserts, and then rebuild them afterwards. In that way, the inserts and bulk copies go faster, since the nonclustered indexes do not have to be updated with every insert. For more information, see “Rebuilding Indexes” on page 9-18.

10 Managing Statistics to Improve Performance

Accurate statistics are essential to the query optimization. In some cases, adding statistics for columns that are not leading index keys also improves query performance. This chapter explains how and when to use the commands that manage statistics.

This chapter contains the following sections:

The Importance of Statistics

Adaptive Server's cost-based optimizer uses statistics about the tables, indexes, and columns named in a query to estimate query costs. It chooses the access method that the optimizer determines has the least cost. But this cost estimate cannot be accurate if statistics are not accurate. Some statistics, such as the number of pages or rows in a table, are updated during query processing. Other statistics, such as the histograms on columns, are only updated when you run the **update statistics** command or when indexes are created.

If you are having problems with a query performing slowly, and seek help from Technical Support or a Sybase news group on the Internet, one of the first questions you are likely be asked is "Did you run **update statistics**?" You can use the **optdiag** command to see the time **update statistics** was last run for each column on which statistics exist:

```
Last update of column statistics: Aug 31 1999  4:14:17:180PM
```

Another command you may need for statistics maintenance is **delete statistics**. Dropping an index does not drop the statistics for that index. If the distribution of keys in the columns changes after the index is dropped, but the statistics are still used for some queries, the outdated statistics can affect query plans.

When Do Statistics Need to Be Updated?

The **update statistics** commands update the column-related statistics such as histograms and densities. So statistics need to be updated on those columns where the distribution of keys in the index changes in ways that affect the use of indexes for your queries.

Running the **update statistics** commands requires system resources. Like other maintenance tasks, it should be scheduled at times when load on the server is light. In particular, **update statistics** requires table scans or leaf-level scans of indexes, may increase I/O contention, may use the CPU to perform sorts, and uses the data and procedure caches. Use of these resources can adversely affect queries running on the server if you run **update statistics** at times when usage is high. In addition, some **update statistics** commands require shared locks, which can block updates. See “Scan Types, Sort Requirements, and Locking During update statistics” on page 10-8 for more information.

Adding Statistics for Unindexed Columns

When you create an index, a histogram is generated for the leading column in the index. Examples in earlier chapters have shown how statistics for other columns can increase the accuracy of optimizer statistics. For example, see “Using Statistics on Multiple Search Arguments” on page 5-16.

You should consider adding statistics for virtually all columns that are frequently used as search arguments, as long as your maintenance schedule allows time to keep these statistics up to date.

In particular, adding statistics for minor columns of composite indexes can greatly improve cost estimates when those columns are used in search arguments or joins along with the leading index key.

The *update statistics* Commands

The **update statistics** commands create statistics, if there are no statistics for a particular column, or replaces existing statistics if they already exist. The statistics are stored in the system tables *systabstats* and *sysstatistics*. The syntax is:

```
update statistics table_name
       [ [index_name] | [( column_list ) ] ]
       [using step values ]
       [with consumers = consumers ]
update index statistics table_name [index_name]
       [using step values ]
       [with consumers = consumers ]
update all statistics table_name
```

The effects of the commands and their parameters are:

- For **update statistics**:
  - *table_name* – Generates statistics for the leading column in each index on the table.
  - *table_name index_name* – Generates statistics for all columns of the index.
  - *table_name* (*column_name*) – Generates statistics for only this column.
  - *table_name* (*column_name, column_name...*) – Generates a histogram for the leading column in the set, and multicolumn density values for the prefix subsets.
- For **update index statistics**:
  - *table_name* – Generates statistics for all columns in all indexes on the table.
  - *table_name index_name* – Generates statistics for all columns in this index.
- For **update all statistics**:
  - *table_name* – Generates statistics for all columns of a table.

Column Statistics and Statistics Maintenance

Histograms are kept on a per-column basis, rather than on a per-index basis. This has certain implications for managing statistics:

- If a column appears in more than one index, **update statistics**, **update index statistics** or **create index** updates the histogram for the column and the density statistics for all prefix subsets. **update all statistics** updates histograms for all columns in a table.
- Dropping an index does not drop the statistics for the index, since the optimizer can use column-level statistics to estimate costs, even when no index exists. If you want to remove the statistics

after dropping an index, you must explicitly delete them with **delete statistics**. If the statistics are useful to the optimizer and you want to keep the statistics without having an index, you need to use **update statistics**, specifying the column name, for indexes where the distribution of key values changes over time.

- Truncating a table does not delete the column-level statistics in *sysstatistics*. In many cases, tables are truncated and the same data is reloaded. Since **truncate table** does not delete the column-level statistics, there is no need to run **update statistics** after the table is reloaded, if the data is the same. If you reload the table with data that has a different distribution of key values, you need to run **update statistics**.
- You can drop and re-create indexes without affecting the index statistics, by specifying **0** for the number of steps in the **with statistics** clause to **create index**. This **create index** command does not affect the statistics in *sysstatistics*:

```
create index title_id_ix on titles(title_id)
    with statistics using 0 values
```

  This allows you to re-create an index without overwriting statistics that have been edited with **optdiag**.
- If two users attempt to create an index on the same table, with the same columns, at the same time, one of the commands may fail due to an attempt to enter a duplicate key value in *sysstatistics*.

Creating and Updating Column Statistics

Creating statistics on unindexed columns can improve the performance of many queries. The optimizer can use statistics on any column in a **where** or **having** clause to help estimate the number of rows from a table that match the complete set of query clauses on that table. Adding statistics for the minor columns of indexes and for unindexed columns that are frequently used in search arguments can greatly improve the optimizers estimates.

Maintaining a large number of indexes during data modification can be expensive. Every index for a table must be updated for each insert and delete to the table, and updates can affect one or more indexes. Generating statistics for a column without creating an index gives the optimizer more information to use for estimating the number of pages to be read by a query, without entailing the processing expense of index updates during data modification.

The optimizer can apply statistics for any columns used in a search argument of a **where** or **having** clause and for any column named in a join clause. You need to determine whether the expense of creating and maintaining the statistics on these columns is worth the improvement in query optimization.

The following commands create and maintain statistics:

- **update statistics**, when used with the name of a column, generates statistics for that column without creating an index on it. The optimizer can use these column statistics to more precisely estimate the cost of queries that reference the column.
- **update index statistics**, when used with an index name, creates or updates statistics for all columns in an index. If used with a table name, it updates statistics for all indexed columns.
- **update all statistics** creates or updates statistics for all columns in a table.

Good candidates for column statistics are:

- Columns frequently used as search arguments in **where** and **having** clauses
- Columns included in a composite index, and which are not the leading columns in the index, but which can help estimate the number of data rows that need to be returned by a query. See "How Scan and Filter Selectivity Can Differ" on page 18-15 for information on how additional column statistics can be used in query optimization.

Identifying When Additional Statistics May Be Useful

To determine when additional statistics are useful, run queries using **dbcc traceon(302)** and **statistics io**. If there are significant discrepancies between the "rows to be returned" and I/O estimates displayed by **dbcc traceon(302)** and the actual I/O displayed by **statistics io**, examine these queries for places where additional statistics can improve the estimates. Look especially for the use of default density values for search arguments and join columns. See "Tuning with dbcc traceon" on page 18-1 for more information.

Adding Statistics for a Column with *update statistics*

This command adds statistics for the *price* column in the *titles* table:

```
update statistics titles (price)
```

This command specifies the number of histogram steps for a column:

```
update statistics titles (price)
    using 50 values
```

This command adds a histogram for the *titles.pub_id* column and generates density values for the prefix subsets *pub_id*; *pub_id*, *pubdate*; and *pub_id*, *pubdate*, *title_id*:

```
update statistics titles(pub_id, pubdate, title_id)
```

➤ ***Note***

Running **update statistics** with a table name updates histograms and densities for leading columns for indexes only. It does not update the statistics for unindexed columns. To maintain these statistics, you must run **update statistics** and specify the column name, or run **update all statistics**.

Adding Statistics for Minor Columns with *update index statistics*

To create or update statistics on all columns in an index, use **update index statistics**. The syntax is:

```
update index statistics table_name [index_name]
       [using step values]
       [with consumers = consumers ]
```

Adding Statistics for All Columns with *update all statistics*

To create or update statistics on all columns in a table, use **update all statistics**. The syntax is:

```
update all statistics table_name
```

Choosing Step Numbers for Histograms

By default, each histogram has 20 steps which provides good performance and modeling for columns that have an even distribution of values. A higher number of steps can increase the accuracy of I/O estimates for:

- Columns with a large number of highly duplicated values
- Columns with unequal or skewed distribution of values
- Columns that are queried using leading wildcards in **like** queries

➤ ***Note***

If your database was updated from a pre-11.9 version of the server, the number of steps defaults to the number of steps that were used on the distribution page.

Disadvantages of Too Many Steps

Increasing the number of steps beyond what is needed for good query optimization can hurt Adaptive Server performance, largely due to the amount of space that is required to store and use the statistics. Increasing the number of steps:

- Increases the disk storage space required for *sysstatistics*
- Increases the cache space needed to read statistics during query optimization
- Requires more I/O, if the number of steps is very large

During query optimization, histograms use space borrowed from the procedure cache. This space is released as soon as the query is optimized.

Choosing a Step Number

See “Choosing the Number of Steps for Highly Duplicated Values” on page 19-21 for more information.

For example, if your table has 5000 rows, and one value in the column that has only one matching row, you may need to request 5000 steps to get a histogram that includes a frequency cell for every distinct value. The actual number of steps is not 5000; it is either the number of distinct values plus one (for dense frequency cells) or twice the number of values plus one (for sparse frequency cells).

Scan Types, Sort Requirements, and Locking During *update statistics*

Table 10-1 shows the types of scans performed during **update statistics**, the types of locks acquired, and when sorts are needed.

Table 10-1: Scans, sorts, and locking during update statistics

update statistics Specifying	Scans and Sorts Performed	Locking
Table name		
Allpages-locked table	Table scan, plus a leaf-level scan of each nonclustered index	Level 1; shared intent table lock, shared lock on current page
Data-only-locked table	Table scan, plus a leaf-level scan of each nonclustered index and the clustered index, if one exists	Level 0; dirty reads
Table name and clustered index name		
Allpages-locked table	Table scan	Level 1; shared intent table lock, shared lock on current page
Data-only-locked table	Leaf level index scan	Level 0; dirty reads
Table name and nonclustered index name		
Allpages-locked table	Leaf level index scan	Level 1; shared intent table lock, shared lock on current page
Data-only-locked table	Leaf level index scan	Level 0; dirty reads
Table name and column name		
Allpages-locked table	Table scan; creates a worktable and sorts the worktable	Level 1; shared intent table lock, shared lock on current page
Data-only-locked table	Table scan; creates a worktable and sorts the worktable	Level 0; dirty reads

Sorts for Unindexed or Nonleading Columns

For unindexed columns and columns that are not the leading columns in indexes, Adaptive Server performs a serial table scan, copying the column values into a worktable, and then sorts the worktable in order to build the histogram. The sort is performed in serial, unless the **with consumers** clause is specified. See Chapter 13, “Parallel Sorting,”, for information on parallel sort configuration requirements.

Locking, Scans, and Sorts During *update index statistics*

The **update index statistics** command generates a series of update statistics operations that use the same locking, scanning, and sorting as the equivalent index-level and column-level command. For example, if the *salesdetail* table has a nonclustered index named *sales_det_ix* on *salesdetail(stor_id, ord_num, title_id)*, this command:

```
update index statistics salesdetail
```

performs these **update statistics** operations:

```
update statistics salesdetail sales_det_ix
update statistics salesdetail (ord_num)
update statistics salesdetail (title_id)
```

Locking, Scans and Sorts During *update all statistics*

The **update all statistics** commands generates a series of **update statistics** operations for each index on the table, followed by a series of **update statistics** operations for all unindexed columns, followed by an **update partition statistics** operation.

Using the *with consumers* Clause

The **with consumers** clause for **update statistics** is designed for use on partitioned tables on RAID devices, which appear to Adaptive Server as a single I/O device, but which are capable of producing the high throughput required for parallel sorting. Chapter 13, “Parallel Sorting,” for more information.

Reducing *update statistics* Impact on Concurrent Processes

Since **update statistics** uses dirty reads (transaction isolation level 0) for data-only locked tables, it can be run while other tasks are active on the server, and does not block access to tables and indexes. Updating statistics for leading columns in indexes requires only a leaf-level scan of the index, and does not require a sort, so updating statistics for these columns does not affect concurrent performance very much.

However, updating statistics for unindexed and nonleading columns, which require a table scan, worktable, and sort can affect concurrent processing.

- Sorts are CPU intensive. Use a serial sort, or a small number of worker processes if you want to minimize CPU utilization. Alternatively, you can use execution classes to set the priority for **update statistics**. See Chapter 37, "How Adaptive Server Uses Engines and CPUs,".
- The cache space required for merging sort runs is taken from the data cache, and some procedure cache space is also required. Setting the **number of sort buffers** to a low value reduces the space used in the buffer cache. If **number of sort buffers** is set to a large value, it takes more space from the data cache, and may also cause stored procedures to be flushed from the procedure cache, since procedure cache space is used while merging sorted values.

Creating the worktables for sorts also uses space in *tempdb*.

Using the *delete statistics* Command

In pre-11.9 versions of SQL Server and Adaptive Server, dropping an index removes the distribution page for the index. In version 11.9.2, maintaining column-level statistics is under explicit user control, and the optimizer can use column-level statistics even when an index does not exist. The **delete statistics** command allows you to drop statistics for specific columns.

If you create an index and then decide to drop it because it is not useful for data access, or because of the cost of index maintenance during data modifications, you need to determine:

- Whether the statistics on the index are useful to the optimizer.
- Whether the distribution of key values in the columns for this index are subject to change over time as rows are inserted and deleted. If the distribution of key values changes, you need to run **update statistics** periodically to maintain useful statistics.

This command deletes the statistics for the *price* column in the *titles* table:

```
delete statistics titles(price)
```

➤ ***Note***

The **delete statistic**s command, when used with a table name, removes all statistics for a table, even where indexes exist. You must run **update statistics** on the table to restore the statistics for the index.

When Row Counts May Be Inaccurate

Row count values for the number of rows, number of forwarded rows, and number of deleted rows may be inaccurate, especially if query processing includes many rollback commands. If workloads are extremely heavy, and the housekeeper task does not run often, these statistics are more likely to be inaccurate. Running **update statistics** corrects these counts in *systabstats*. Running **dbcc checktable** or **dbcc checkdb** updates these values in memory. When the housekeeper task runs, or when you execute **sp_flushstats**, these values are saved in *systabstats*.

➤ ***Note***

The configuration parameter **housekeeper free write percent** must be set to 1 or greater to enable housekeeper statistics flushing.

Parallel Query Concepts and Tuning

11 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:

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 12, "Parallel Query Optimization."
- To understand parallel sorting topics, see Chapter 13, "Parallel Sorting."
- For information on object placement for parallel performance, see "Partitioning Tables for Performance" on page 33-14.
- For information about locking behavior during parallel query processing, see Chapter 26, "Locking in Adaptive Server."
- For information on **showplan** messages, see "showplan Messages for Parallel Queries" on page 17-42.
- To understand how Adaptive Server uses multiple engines, see Chapter 37, "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 are processed 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 that use merge joins can use parallel processing for scanning tables and for performing the sort and merge steps
- **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**), 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.

Adaptive Server's Worker Process Model

Adaptive Server uses a **coordinating process** and multiple **worker processes** to execute queries in parallel. 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 11-1 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. 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 the results produced by all the worker processes.

The serial client shown in the lower-right corner of Figure 11-1 submits a query that is processed serially.

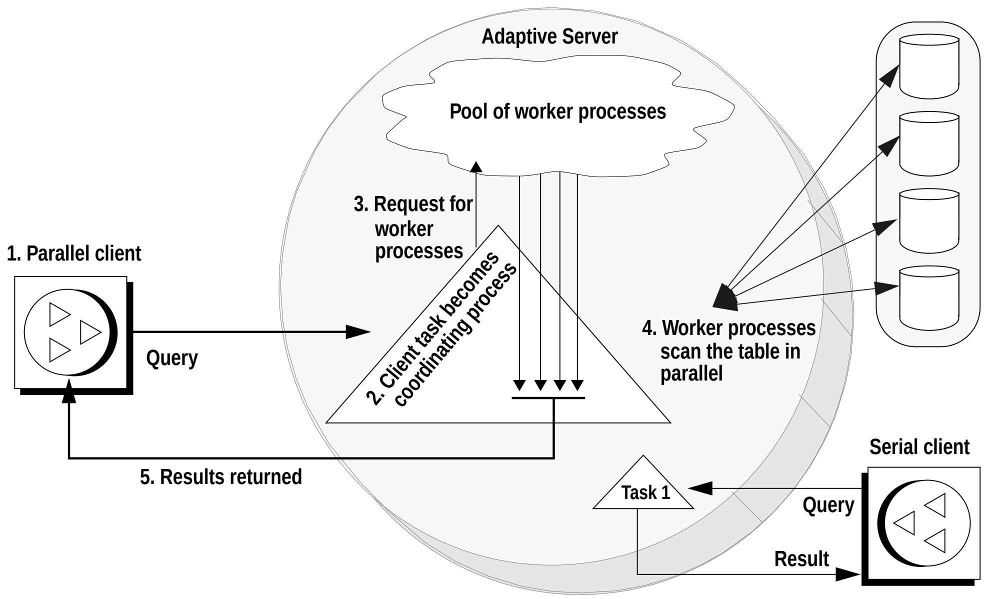


Figure 11-1: Worker process model

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 11-2 shows how parallel query processing reduces response time over the same query running in serial. 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.

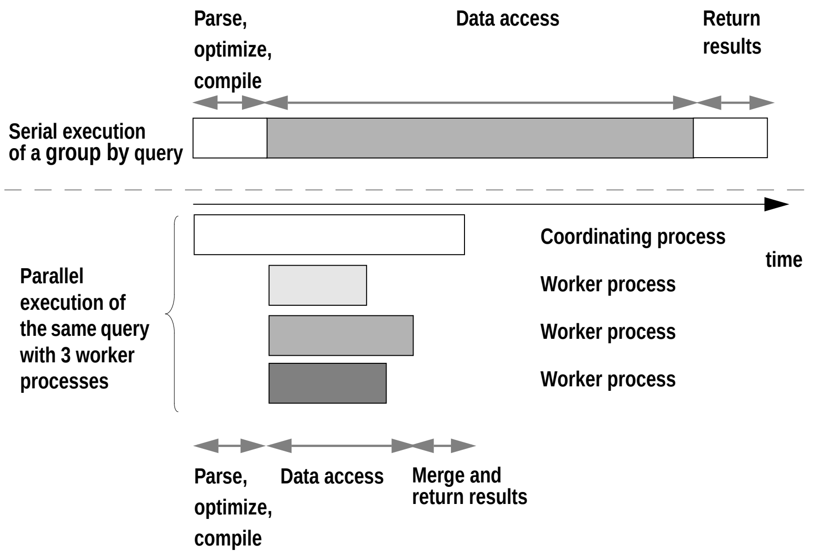


Figure 11-2: Relative execution times for serial and parallel query execution

The total amount of work performed by the query running in parallel is greater 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 the results 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 and that do not use clauses that do not require a final 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 17-42 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 13, "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 step 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 11-26.

Types of Parallel Data Access

Adaptive Server accesses data in parallel in different ways, depending configuration parameter settings, table partitioning, and the availability of indexes. The optimizer may choose a mix of serial and parallel methods for 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
- Range-based scans during merge joins

The following sections describe some of the methods. For more examples, see Chapter 12, "Parallel Query Optimization."

Figure 11-3 shows a scan on an allpages-locked 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.

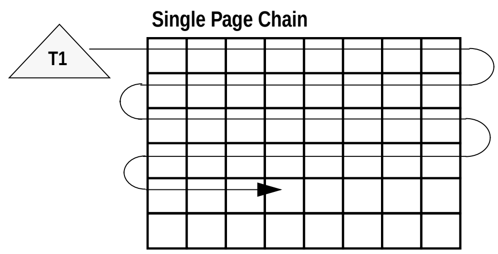


Figure 11-3: A serial task scans data pages

Hash-Based Table Scans

Figure 11-4 shows how three worker processes divide the work of accessing data pages from an allpages-locked table during a hash-based table scan. Each worker process performs a logical I/O on every page, 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 are multiple engines, some of the worker processes could be running simultaneously.

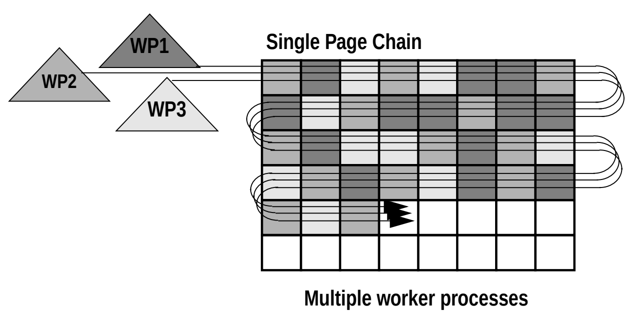


Figure 11-4: Worker processes scan an unpartitioned table

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. For data-only-locked tables, hash-based table scans hash either on the extent ID or the allocation page ID, so that only a single worker process scans a page, and logical I/O does not increase.

Partition-Based Scans

Figure 11-5 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 waiting for I/O or waiting 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.

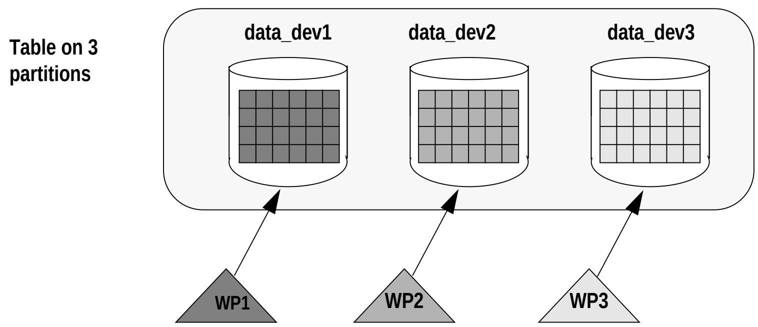


Figure 11-5: Multiple worker processes access multiple partitions

Hash-Based Index Scans

Figure 11-6 shows a hash-based index scan. Hash-based index scans can be performed using nonclustered indexes or clustered indexes on data-only-locked tables. 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.

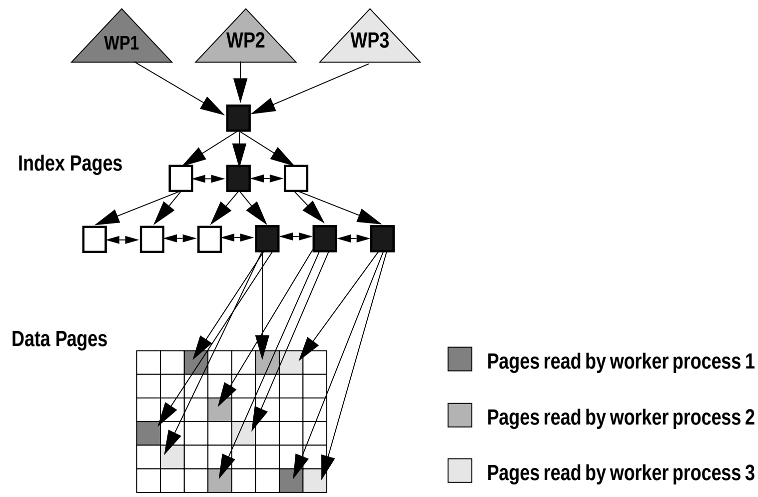


Figure 11-6: Hash-based, nonclustered index scan

Parallel Processing for Two Tables in a Join

Figure 11-7 shows a nested-loop join query performing a partition-based scan on a table with three partitions, and a hash-based index scan, with two worker processes on the second table. When parallel access methods are used on more than one table in a nested-loop 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.

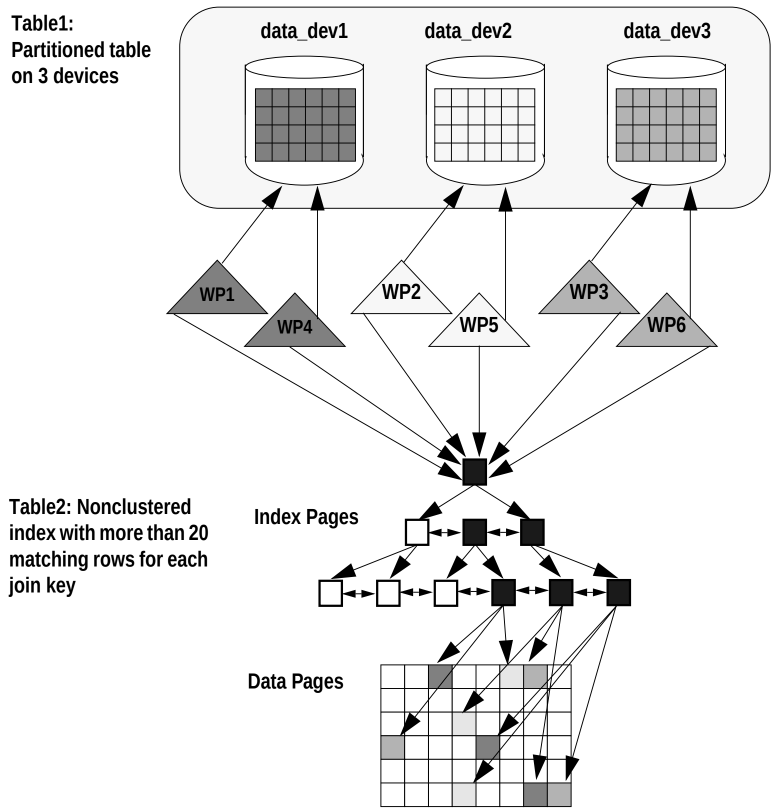


Figure 11-7: Join query using different parallel access methods on each table

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 11-7:

```
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. For the query shown in Figure 11-7, it reports:

```
Executed in parallel by coordinating process and 6
worker processes.
```

See "showplan Messages for Parallel Queries" on page 17-42 for more information and Chapter 12, "Parallel Query Optimization," for additional examples.

Controlling the Degree of Parallelism

A parallel query's **degree of parallelism** is the number of worker processes used to execute the query. This number depends on several factors, including:

- The values to which of the parallel configuration parameters or the session-level limits, (see Table 11-1 and Table 11-2)
- The number of partitions on a table (for partition-based scans)
- The level of parallelism suggested by the optimizer
- 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 **sp_configure** with parameters shown in Table 11-1. Only a System Administrator can use **sp_configure**.
- For a session – Using **set** with the parameters shown in Table 11-2. All users can run **set**; it can also be included in stored procedures.
- In a **select** query – Using the **parallel** clause, as shown in "Controlling Parallelism for a Query" on page 11-16.

Configuration Parameters for Controlling Parallelism

The configuration parameters that give you control over the degree of parallelism server-wide are shown in Table 11-1.

Table 11-1: 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

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 17-102 in the *System Administration Guide*.

When you change **max parallel degree** or **max scan parallel degree**, all query plans in cache are invalidated, so the next execution of any stored procedure or trigger recompiles the plan and uses the new values.

How Limits Apply to Query Plans

When queries are optimized, 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 nested-loop join queries, where parallel access methods are used on more than one table
  - The number of worker processes used for the merge and sort steps in merge joins

- 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 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 nested-loop 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 partition- 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 merge joins:

- For a full-merge join, 10 worker processes scan the base tables (unless these are fewer than 10 distinct values on the join keys); the number of partitions on the tables is not considered.
- For a merge join that scans a table and selects rows into a worktable:

- The scan that precedes the merge join may be performed in serial or in parallel. The degree of parallelism is determined in the usual way for such a query.
- For the merge, 10 worker processes are used unless there are fewer distinct values in the join key.
- For the sort, up to 10 worker processes can be used.

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 13-6.

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:

```
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 11-2.

Table 11-2: set options for parallel execution tuning

Parameter	Function
parallel_degree	Sets the maximum number of worker processes for a query in a session, stored procedure, or trigger. Overrides the **max parallel degree** configuration parameter, but must be less than or equal to the value of **max parallel degree**.
scan_parallel_degree	Sets the maximum number of worker processes for a hash-based scan during a specific session, stored procedure, or trigger. Overrides the **max scan parallel degree** configuration parameter but must be less than or equal to the value of **max scan parallel degree**.

If you specify a value that is too large for **set** 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 executed 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 use a partition-based scan.

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)
```

See “Suggesting a Degree of Parallelism for a Query” on page 20-14 for more information.

Worker Process Availability and Query Execution

At runtime, 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 **runtime adjustment**, and it can result in serial execution of the query.

A runtime adjustment now and then probably indicates an occasional, momentary bottleneck. Frequent runtime adjustments indicate that the system may not be configured with enough worker processes for the workload. See “Runtime Adjustment of Worker Processes” on page 12-28 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 12-30 for more information.

Runtime 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 13-11 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 39-17 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 in Chapter 33, "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 11-8 shows a scenario for creating a new partitioned table.

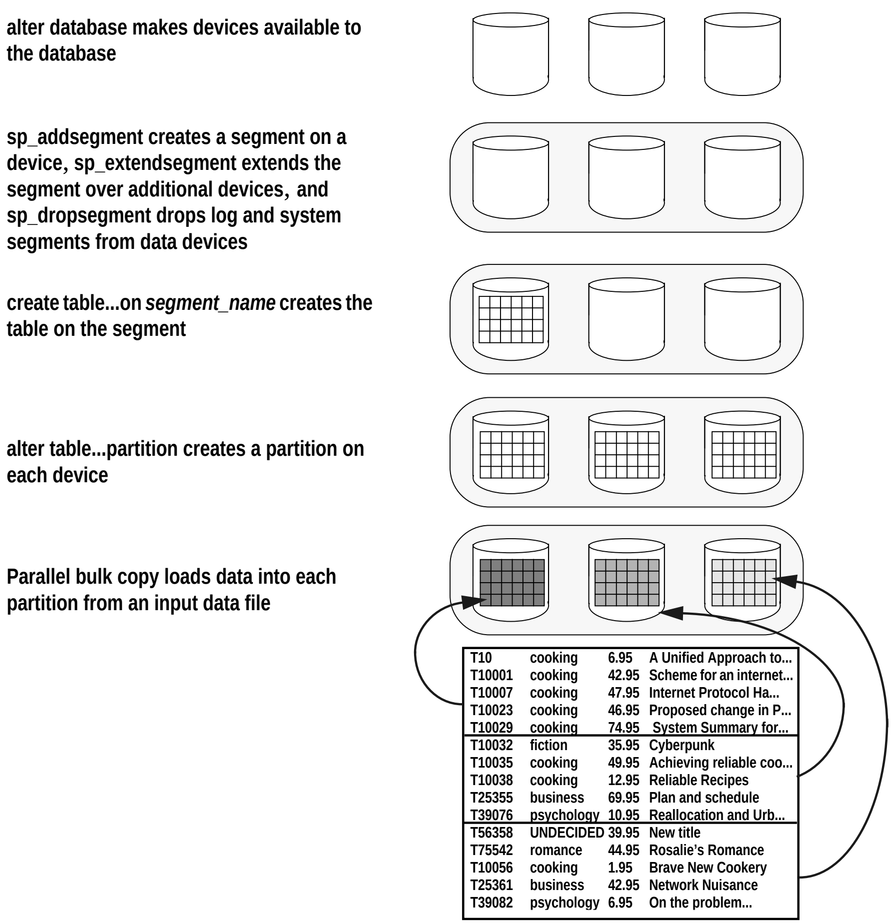


Figure 11-8: Steps for creating and loading a new partitioned table

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 both serial and parallel queries, can be degraded. See Chapter 38, “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 runtime 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 11-3 shows the CPU scaling.

Table 11-3: 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 Table 11-3 to provide illustrate how to use 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 11-3, 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 11-3.

- 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 11-3 had 30 partitions available. Following the suggestions in the hardware guidelines of one CPU for each five devices suggests using six 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 11-20.

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 can be 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 can 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. 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; 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. Queries hold shared locks on data pages during the scans and cause data modifications to wait for the shared locks to be released. You may need to restrict queries with large result sets 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 query, because the rollback decision for deadlocks is based on the accumulated CPU time of the processes involved. See “Deadlocks and Concurrency” on page 28-5 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 differences in result set ordering. Each parallel scan behaves 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, 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 accessing the same data may return different results 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, because 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 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

12 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. Parallel query optimization is an automatic process, and the optimized query plans created by Adaptive Server generally yield the best response time 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 expect. 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?

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 earlier chapters. In addition to the costing performed for serial query optimization, parallel optimization analyzes the cost of parallel access methods for each combination of join orders, join types, and indexes. The optimizer can choose any combination of serial and parallel access methods to create the fastest query plan.

Optimizing for Response Time vs. Total Work

Serial query optimization selects the query 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, a parallel query plan 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, serial query optimization improves performance by minimizing the use of server resources, but parallel query optimization improves performance for individual queries by fully utilizing available resources to get the fastest response time.

When Is Parallel Query Optimization Performed?

The optimizer considers parallel query plans only when Adaptive Server and the current session are properly configured for parallelism, as described in "Controlling the Degree of Parallelism" on page 11-11.

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 considers serial access methods; the parallel access methods described in this chapter are not considered.

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 a table. This restriction applies whether or not an index is used to access a table's data. When Adaptive Server must scan fewer than 20 data pages, the optimizer considers only serial table and index scans and does not consider parallel optimization.

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 "Configuration Parameters for Controlling Parallelism" on page 11-12 for information on configuring Adaptive Server. See "Commands for Partitioning Tables" on page 33-21 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. Parallel access methods fall into these general categories:

- **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, using either table scans or index scans. 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.
- **Range-based** methods provide parallel access during merge joins on partitioned tables and unpartitioned tables, including worktables created for sorting and merging, and via indexes. The partitioning on the tables is not considered when choosing the degree of parallelism, so it is not distributed over physical devices or controllers. Worker processes can 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 all pages in the partition. Figure 12-1 illustrates a parallel partition scan.

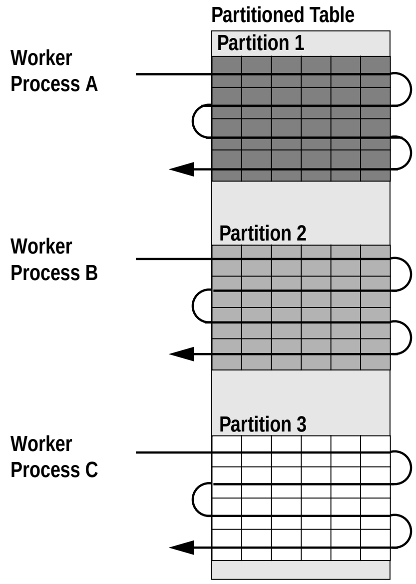


Figure 12-1: Parallel partition scan

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 cannot 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 logical and physical I/Os performed by any one worker process in the scan. In other words, the cost of this access method equals the I/O required to read all 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 logical and 500 physical I/Os (assuming 2K I/O for the physical I/O). In contrast, the cost of a serial scan of this table is 1000 logical and physical I/Os.

Parallel Clustered Index Partition Scan (Allpages-Locked Tables)

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. This method can be used only on allpages-locked tables.

One worker process is assigned to each partition in the table. Each worker process accesses data pages in the partition, using one of two methods, depending on the range of key values accessed by the process. When a partitioned table has a clustered index, rows are assigned to partitions based on the clustered index key.

Figure 12-2 shows a clustered index partition 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 search argument.

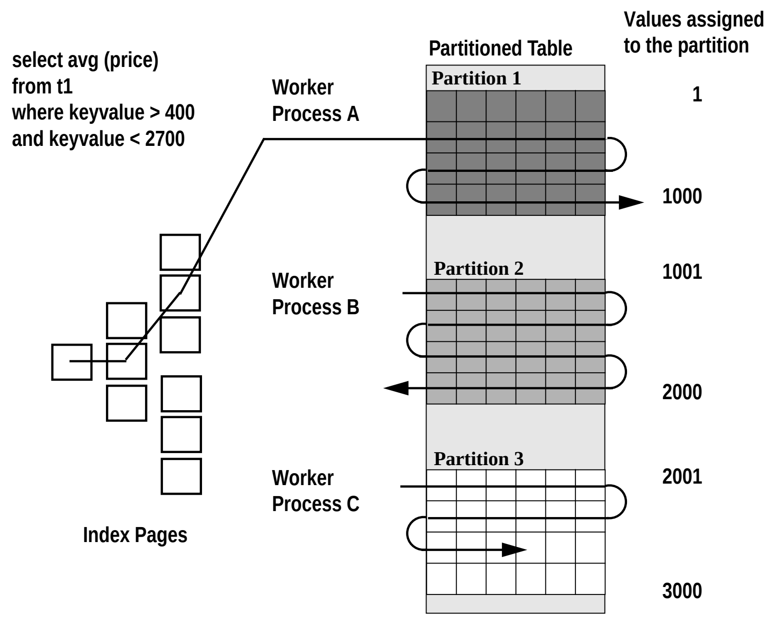


Figure 12-2: Parallel clustered index partition scan

Requirements for Consideration

The optimizer considers a clustered index partition scan only when:

- The query accesses at least 20 data pages of the table.
- The table is partitioned and uses allpages locking.
- 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, 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, 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 Hash-Based Table Scan

Parallel hash-based table scans are performed slightly differently, depending on the locking scheme of the table.

Hash-Based Table Scans on Allpages-Locked Tables

In a hash-based table scan on an allpages-locked table, multiple worker processes scan a single chain of data pages in a table simultaneously. 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 12-3 illustrates the hash-based table scan.

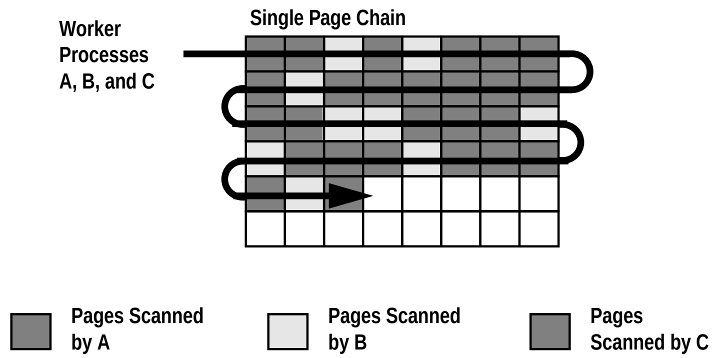


Figure 12-3: Parallel hash-based table scan on an allpages-locked table

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.

Hash-Based Table Scans on Data-Only-Locked Tables

A hash-based scan on a data-only-locked table hashes on either the extent number or the allocation page number, rather than hashing on the page number. The choice of whether to hash on the allocation page or the extent number is a cost-based decision made by the optimizer. Both methods can reduce the cost of performing parallel queries on unpartitioned tables. Queries that choose a serial scan on an allpages-locked table may use one of the new hash-based scan methods if the table is converted to data-only locking.

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 clustered indexes or for 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 optimizer computes the cost of a hash-based table scan as the total number of logical and physical I/Os required to scan the table.

For an allpages-locked table, the physical I/O cost is approximately the same as for a serial table scan. The logical cost is the number of pages to be read multiplied by the number of worker processes. The cost per worker process is one logical I/O for each page in the table, and approximately 1/*N* physical I/Os, with *N* being the number of worker processes.

For a data-only-locked table, this is approximately the same cost applied to a serial table scan, with the physical and logical I/O divided evenly between the worker processes.

Parallel Hash-Based Index Scan

An index hash-based scan can be performed using either a nonclustered index or a clustered index on a data-only-locked table. To perform the scan:

- All worker processes traverse the higher index levels.
- All worker processes scan the leaf-level index pages.

For data-only-locked tables, the worker processes scanning the leaf level hash on the page ID for each row, and scan the matching data pages.

For allpages-locked tables, a hash-based index scan is performed in one of two ways, depending on whether the table is a heap table or 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 heap table, the scan hashes on the page ID.

Figure 12-4 illustrates a nonclustered index hash-based scan on a heap table with two worker processes.

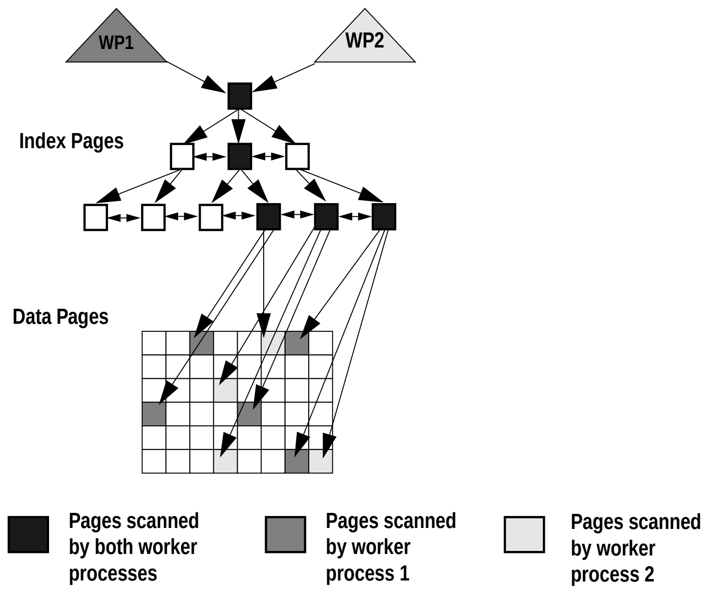


Figure 12-4: Nonclustered index hash-based scan

Cost Model and Requirements

The cost model of a nonclustered index scan uses the formula:

Scan Cost = Number of index levels
+ Number of leaf pages / pages per IO
+ (Number of data pages / pages per IO) / number of worker processes

The optimizer considers a hash-based index scan for any tables in a query that have useful nonclustered indexes, and for data-only-locked tables with clustered indexes. The query must also access at least 20 data pages of the table.

➤ ***Note***

If a nonclustered index covers the result of a query, 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.

Parallel Range-Based Scans

Parallel range-based scans are used for the merge process in merge joins.

When two tables are merged in parallel, each worker process is assigned a range of values to merge. The range is determined using histogram statistics or sampling. When a histogram exists for at least one of the join columns, it is used to partition the ranges so that each worker process operates on approximately the same number of rows. If neither join column has a histogram, sampling similar to that performed for other parallel sort operations determines the range of values to be merged by each worker process.

Figure 12-5 shows a parallel right-merge join. In this case:

- A right-merge join is used. Table1, the outer table, is scanned into a worktable and sorted, then merged with the inner table. These worker processes are deallocated at the end of this step.
- The outer table has two partitions, so two worker processes are used to perform a parallel partition scan.
- The inner table has a nonclustered index on the join key. **max parallel degree** is set to 3, so 3 worker processes are used.

Requirements for Consideration

The optimizer considers parallel merge joins when the configuration parameter **enable merge joins** is set to 1 and the table accesses more than 20 data pages from the outer table in the merge join.

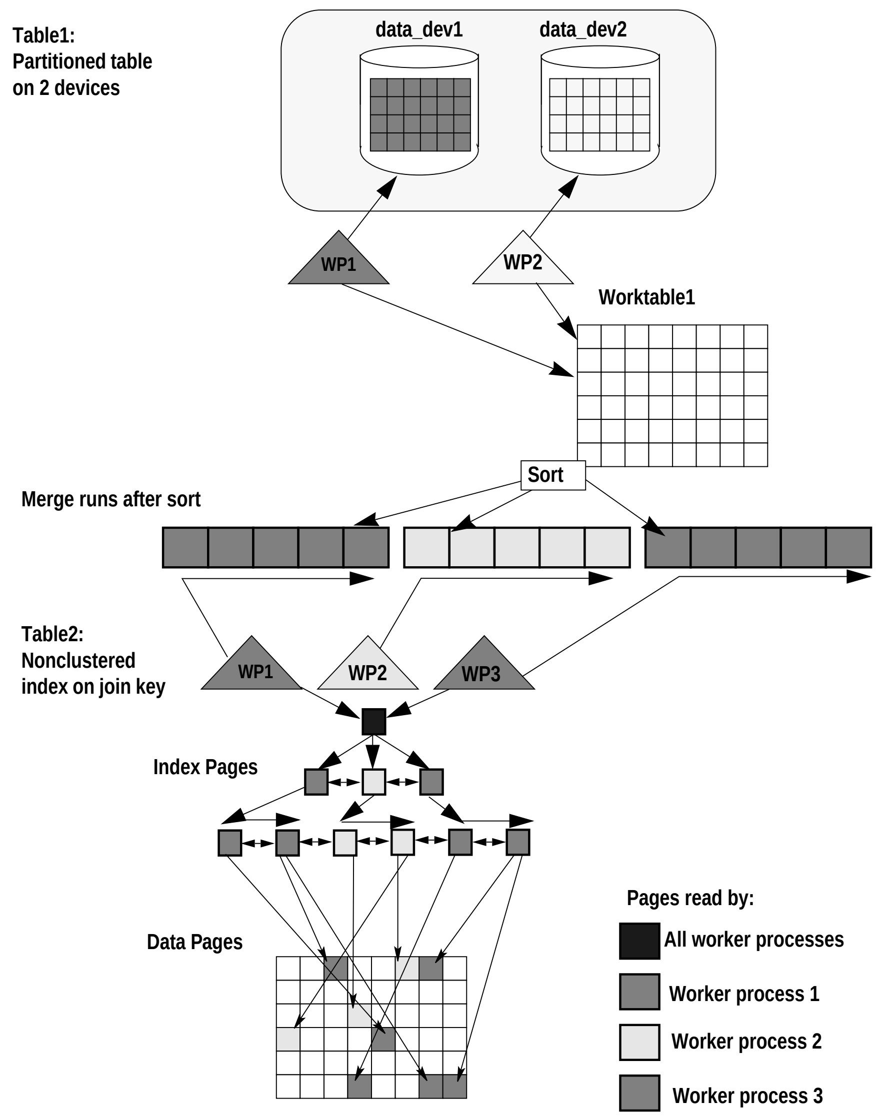


Figure 12-5: A parallel right-merge join

Additional Parallel Strategies

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 12-24 for examples of queries that can benefit from the use of partitioned worktables.

Parallel Sorting

Parallel sorting 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 optimizer does not directly optimize or control the execution of a parallel sort. See "Parallel Query Examples" on page 12-24 for examples of queries that can benefit from the parallel sorting strategy. Also, see "Overview of the Parallel Sorting Strategy" on page 13-3 for a detailed explanation of how Adaptive Server executes a sort in parallel.

Summary of Parallel Access Methods

Table 12-1summarizes 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 12-1: Parallel access method summary

Parallel Method	Major Cost Factors	Requirements for Consideration	Competing Serial Methods
Partition-based 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 <= 2 * 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 * number of pages in average-sized partition, then: Average number of pages in a partition	Partitioned table with a useful clustered index; allpages locking only	Serial index scan
Hash-based index 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 or a data-only-locked table with a clustered index	Serial index scan
Range-based scan	Number of pages to be accessed in both tables / number of worker processes, plus any sort costs	Any table in a join eligible for merge join consideration	Serial merge, nested-loop join

Selecting Parallel Access Methods

For a given table in a query, the optimizer first evaluates the available indexes and partitions to determine which access methods it can use to scan the table's data. For any query that involves a join, Adaptive Server considers a range-based merge join, and considers using a parallel merge join if parallel query processing is enabled. The use of a range-based scan does not depend on table partitioning, and range-based scans can be performed using clustered and nonclustered indexes. They are considered, and are very likely to be used, on tables that have no useful index on the join key.

Table 12-2 shows the other parallel 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 12-2: Determining applicable partition or hash-based access methods

	No Useful Index	Useful Clustered Index	Useful Index (nonclustered or clustered on data-only-locked table)
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 12-22.

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 on both the upper limit to the degree of parallelism for the query and on the level of parallelism suggested by the optimizer.

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 11-12 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 may be less than this maximum. For partition-based scans, 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.

Worker Processes for Partition-Based Scans

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; if a merge join can be used, it may choose a merge join using the **max parallel degree**.

Worker Processes for Hash-Based Scans

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.

Worker Processes for Range-Based Scans

A merge join can use multiple worker processes to perform:

- The scan that selects rows into a worktable, for any merge join that requires a sort
- The worktable sort
- The merge join and subsequent joins in the step
- The range scan of both tables during a full merge join

Worker Process Usage While Creating the Worktable

If a worktable is needed for a merge join, the query step that creates the worktable can use a serial or parallel access method for the scan. The number of worker processes for this step is determined by the usual methods for selecting the number of worker processes for a query. The query that selects the rows into the worktable can be a single-table query or a join performing a nested-loop or merge join, or a combination of nested-loops joins and a merge join.

Parallel Sorting for Merge-Join Worktables

Parallel sorting is used when the number of pages in the worktable to be sorted is eight times the value of the configuration parameter **number of sort buffers**. See Chapter 13, “Parallel Sorting,” for more information about parallel sorting.

Number of Merge Threads

For the merge step, the number of merge threads is set to **max parallel degree**, unless the number of distinct values is smaller than **max parallel degree**. If the number of values to be merged is smaller than the **max parallel degree**, the task uses one worker process per value, with each worker process merging one value. If the tables being merged have different numbers of distinct values, the lower number determines the number of worker processes to be used. The formula is:

Worker processes = min (max pll degree, min(t1_uniq_vals, t2_uniq_vals))

When there is only one distinct value on the join column, or there is an equality search argument on a join column, the merge step is performed in serial mode. If a merge join is used for this query, the merge is performed in serial mode:

```
select * from t1, t2
where t1.c1 = t2.c1
and t1.c1 = 10
```

Total Worker Process Usage for Merge Joins

A merge join can use up to **max parallel degree** threads for the merge step and up to **max parallel degree** threads can be used for each sort. A merge that performs a parallel sort may use up to 2***max parallel degree** threads. Worker processes used for sorts are released when the sort completes.

Degree of Parallelism for Nested-Loop Joins

For individual tables in a nested-loop join, the optimizer computes the degree of parallelism using the same rules described in "Optimized Degree of Parallelism" on page 12-17. 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 12-6 illustrates this rule for two tables in a join where the outer table has three partitions and the inner table has two partitions. If the optimizer determines that partition-based access methods are to be used on each table, then the query requires a total of six worker processes to execute the join. Each of the six worker processes scans one partition of the outer table and one partition of the inner table to process the join condition.

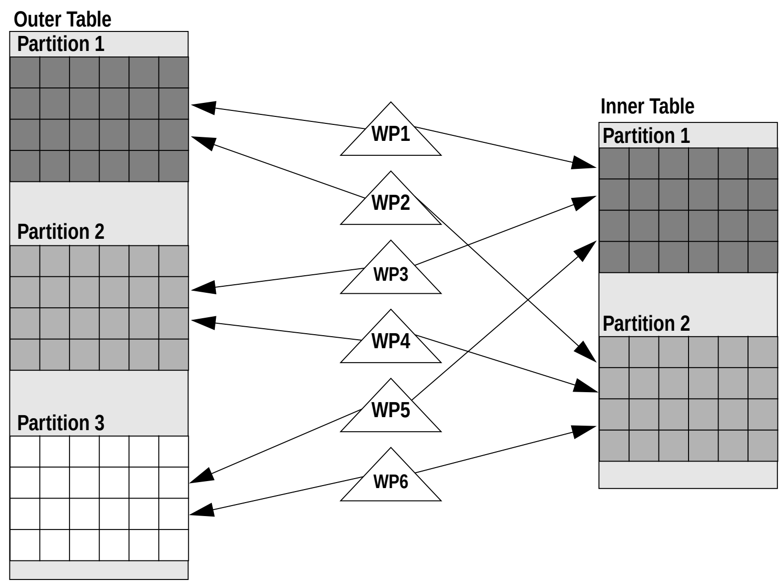


Figure 12-6: Worker process usage for a nested-loop join

In Figure 12-6, if the optimizer chose to scan the inner table using a serial access method, only three 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 two tables in a query with scan degrees of *m* and *n* respectively, the potential degrees of parallelism for a nested-loop 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 accesses 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:

- A merge join.
- The reformatting strategy, if reformatting is a cheaper alternative.
- A partitioned-based scan plus a hash-based index scan, when a join returns rows from 20 or more data pages. See Figure 11-7 on page 11-10 for an illustration.

Computing the Degree of Parallelism for Nested-Loop 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 12-17.
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:

Remaining worker processes = **max parallel degree**/ Worker processes for 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 "Degree of Parallelism for Nested-Loop Joins" on page 12-19 for examples of this process.

Parallel Queries and Existence Joins

Adaptive Server imposes an additional restriction for subqueries processed as existence joins. For these queries, only the number of partitions in the outer table determines the degree of parallelism. There are only as many worker processes as there are partitions in the outer table. The inner table in such a query is always accessed serially. This restriction does not apply to subqueries that are flattened into regular joins.

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.

A Single-Table Query

For a single-table query on a heap table with 6 partitions and no useful nonclustered index, the optimizer costs the following access methods:

- A parallel partition scan using 6 worker processes
- A serial table scan using a single process

If **max parallel degree** is set to 5 worker processes, then the optimizer does not consider the partition scan for a table with 6 partitions.

A Query with a Join

The situation changes if the query involves a join. If **max parallel degree** is set to 10 worker processes, the query involves a join, and a table with 6 partitions is the outer table in the query, then the optimizer considers the following access methods:

- A partition scan using 6 worker processes
- A hash-based table scan using 3 worker processes

- A merge join using 10 worker processes
- A serial scan using a single process

If **max scan parallel degree** is set to 5 and **max scan parallel degree** is set to 3, then the optimizer considers the following access methods:

- A hash-based table scan using 3 worker processes
- A merge join using 5 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 considers only a merge join as a parallel access method.

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 range scan using 10 worker processes for the merge join
- A serial scan using a single process

If **max scan parallel degree** is set to 1, then the optimizer does not consider the hash-based scan.

See “Single-Table Scans” on page 12-24 for more examples of determining the degree of parallelism for queries.

Table with Clustered Index Example

If the table has a clustered index, the optimizer considers the following parallel access methods when the table uses allpages locking:

- A parallel partition scan or a parallel clustered index scan, if the table is partitioned and **max parallel degree** is set to at least 6
- A range scan, using **max parallel degree** worker processes
- A serial scan

If the table uses data-only-locking, the optimizer considers:

- A parallel partition scan, if the table is partitioned and **max parallel degree** is set to at least 6
- A range scan, using **max parallel degree** worker processes

- A serial scan

Runtime 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 runtime to compensate for the actual number of worker processes that are available. If fewer worker processes are available at runtime 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. “Runtime Adjustment of Worker Processes” on page 12-28 describes the process of adjusting the degree of parallelism at runtime and explains how to determine when these adjustments occur.

Parallel Query Examples

The following sections further explain and provide examples of how Adaptive Server optimizes these types of parallel queries:

- Single-Table Scans
- 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 parallel query optimization involves 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 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
authors	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 12-2 on page 12-16, 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 pages

Cost of serial table scan = # of pages in table = 300 pages

The optimizer chooses to perform a table partition scan at a cost of 90 physical and logical I/Os. Because the table has 5 partitions, the optimizer chooses to use 5 worker processes. The final **showplan** output for this query 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.
     Forward scan.
     Positioning at start of table.
     Executed in parallel with a 5-way partition scan.
     Using I/O Size 16 Kbytes for data pages.
     With LRU Buffer Replacement Strategy for data pages.
     Parallel network buffer merge.
```

Subqueries

When a query contains a subquery, Adaptive Server uses different access methods to reduce the cost of processing the subquery. Parallel optimization depends on the type of subquery and the access methods:

- Materialized subqueries – Parallel query methods are not considered for the materialization step.
- Flattened subqueries – Parallel query optimization is considered only when the subquery is flattened to a regular join. It is not considered for existence joins or other flattening strategies.
- Nested subqueries – Parallel access methods are considered for the outermost query block in a query containing a subquery; the inner, nested queries always execute serially. Although the optimizer considers parallel access methods for only 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.

Each worker process accesses 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 parallel sort manager determines whether to use a serial or parallel sort. See Chapter 13, "Parallel Sorting," for more information about parallel sorting.

union Queries

The optimizer considers parallel access methods for each part of a **union** query separately. Each **select** in a union is optimized separately, so one query can use a parallel plan, another a serial plan, and a third a parallel plan with a different number of worker processes. If a **union** query requires a worktable, then the worktable may also be partitioned and populated in parallel by worker processes.

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 13, "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

select into 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. 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 than the equivalent serial query plan. Therefore, the execution of a parallel **select into** statement takes place using four discrete transactions, rather than the two transactions of a serial **select into** statement. See **select** in the *Adaptive Server Reference Manual* for information about how this affects the database recovery process.

Runtime 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 when 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 generated at runtime and 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**, which allows you to control whether runtime adjustments are allowed.
- Information on runtime 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 five partitions using a partition-based table scan. If only four 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 five worker processes, the adjusted query plan would still use a hash-based access method, but with, at the most, four worker processes.

Evaluating the Effect of Runtime 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 Runtime Adjustments

Adaptive Server provides two mechanisms to help you observe runtime adjustments of query plans. These mechanisms include:

- **set process_limit_action** allows you to abort batches or procedures when runtime adjustments take place or print warnings.
- **showplan** prints an adjusted query plan when runtime 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 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 Runtime 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 runtime adjustment is made, **showplan** displays this message, followed by the adjusted query plan:

```
AN ADJUSTED QUERY PLAN WILL BE USED FOR STATEMENT
1 BECAUSE NOT ENOUGH WORKER PROCESSES ARE
AVAILABLE AT THIS TIME.
```

Adaptive Server does not attempt to execute a query when the **set noexec** is in effect, so runtime plans are never displayed while using this option.

Reducing the Likelihood of Runtime Adjustments

To reduce the number of runtime 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 runtime adjustments for system procedures, 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 Runtime Adjustments with *sp_sysmon*

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 39-17 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 39-19 describes the **sp_sysmon** output that reports on the number of runtime 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 re-run **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, decreases 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 runtime adjustments, using the tools discussed in "Runtime Adjustments to Worker Processes" on page 12-24.
- 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 recommended 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 the 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 three 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 6000, the optimizer assigns a limit of 2000 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 may get an error message saying you have exceeded your I/O resource limit when, according to **showplan** or **statistics io** output, you actually have not. Conversely, one partition may exceed the limit slightly, without the limit taking effect.

See Chapter 18, "Limiting Access to Server Resources," in the *System Administration Guide* for more information about setting resource limits.

13 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 to get the best performance from parallel sorting, and to keep parallel sort resource requirements from interfering with other resource needs.

This chapter contains the following sections.

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 perform merge joins requiring sorts
- 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 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 creating other indexes, serial sorts can be performed if parallel sorts cannot be performed.
  - For sorting worktables, this option must be on in *tempdb*. 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 there are not enough worker processes available. "Worker Process Requirements During Parallel Sorts" on page 13-6 and "Worker Process Requirements for select Query Sorts" on page 13-10.
- 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 13-12.

- 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 13-12.

➤ ***Note***

You cannot use the **dump transaction** command after indexes are created using a parallel sort. You must dump the database. Serial **create index** commands can be recovered, but only 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. Creating a distribution map. For a merge join with statistics on a join column, histogram statistics are used for the distribution map. In other cases, the input table is sampled to build the 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 13-1 depicts a parallel sort of a table with two partitions and two physical devices on its segment.

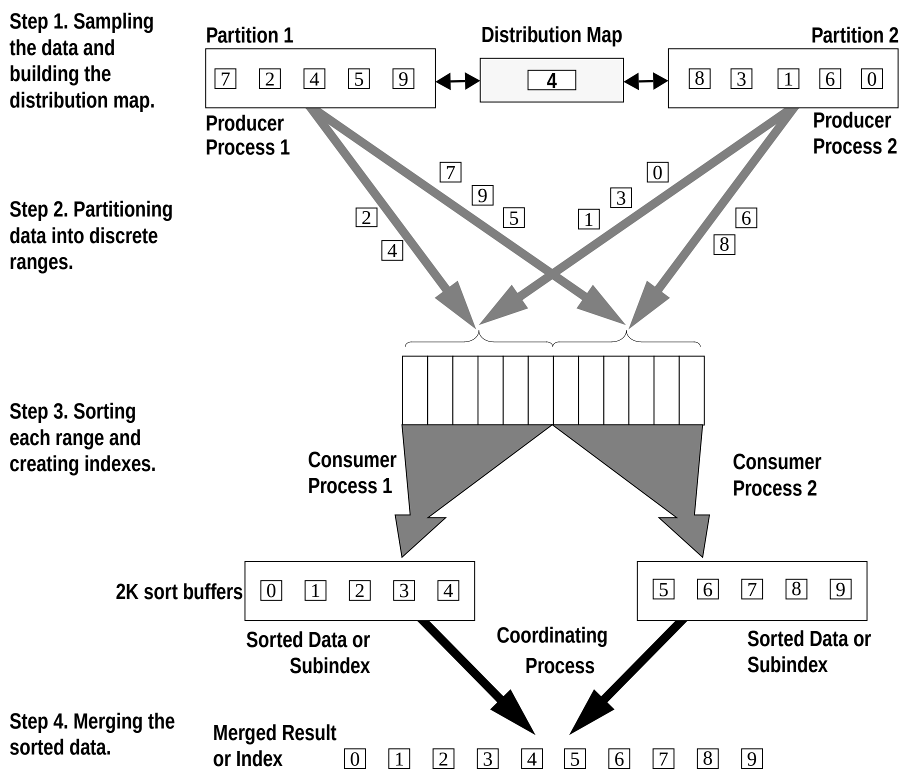


Figure 13-1: Parallel sort strategy

Creating a Distribution Map

As a first step in executing a parallel sort, Adaptive Server creates a distribution map. If the sort is performed as part of a merge join, and there are statistics on the join columns, the histograms are used to build the distribution map. For other sorts, Adaptive Server selects and sorts a random sample of data from 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 is assigned to each range, except the final range in the table. In Figure 13-1, 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 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 13-5.

In Figure 13-1, 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. 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. The consumer process also builds an index, referred to as a subindex, on the sorted data.
- For merge joins, each consumer process writes the ordered rows to a separate set of linked data pages, one for each worker process that will perform the merge.

- For queries, the consumer process simply orders the data in the range from the smallest value to the largest.

Merging Results

After all consumer processes have finished sorting the data for each partitioned range:

- For **create index** commands, the coordinating process merges the subindexes into one final index.
- For merge joins, the worker processes for the merge step perform the merge with the other tables in the merge join.
- For other queries, the coordinating process merges the sort results and returns them to the client.

Configuring Resources for Parallel Sorting

The following sections describe the 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 determine the number of worker processes for most sorts.

Worker Process Requirements During Parallel Sorts

Adaptive Server requires a minimum number of worker processes to perform a parallel sort. If additional worker processes are available, the sort can be performed 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 creating clustered indexes
- Devices, for creating nonclustered indexes
- Threads used to create the worktable and the number of devices in *tempdb*, for merge joins
- Devices in *tempdb*, for other 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; the sort fails if not enough worker processes are available.
- Sorts for nonclustered indexes and sorts for clustered indexes on unpartitioned tables can be performed in serial.
- All sorts for queries can be performed in serial.

The availability of worker processes is determined 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 runtime 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 runtime, 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 11-11 for more information about controlling the server-wide and session-wide limits.

Worker Process Requirements for Creating Indexes

Table 13-1 shows the number of producers and consumers required to create indexes. The **target segment** for a sort is the segment where the index is stored when the **create index** command completes. When you create an index, you can specify the location with the **on** *segment_name* clause. If you do not specify a segment, the index is stored on the *default* segment.

Table 13-1: 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 the **primary key** and **unique** indexes:

```
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

Queries that require worktable sorts have multistep query plans. The determination of the number of worker processes for a worktable sort is made after the scan of the base table completes. 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.

Once the worktable is populated, 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 used in the previous step do not participate in the sort, but remain allocated to the parallel task until the task completes.

Worker Processes for Merge-Join Sorts

For merge joins, one consumer process is assigned for each device in *tempdb*; if there is only one device in *tempdb*, two consumer processes are used. The number of producers depends on the number of partitions in the worktable, and the setting for **max parallel degree**:

- If the worktable is not partitioned, one producer process is used.
- If the number of consumers plus the number of partitions in the worktable is less than or equal to **max parallel degree**, one producer process is allocated for each worktable partition.
- If the number of consumer processes plus the number of partitions in the worktable is greater than **max parallel degree**, one producer process is used.

For Other Worktable Sorts

For all other worktable sorts, the worktable is unpartitioned when the step that created it completes. Worker processes are assigned in the following way:

- If there is only one device in *tempdb*, the sort is performed using two consumers and one producer; otherwise, one consumer process is assigned for each device in *tempdb*, and a single producer process scans the worktable.
- If there are more devices in *tempdb* than the available worker processes when the sort starts, the sort is performed in serial.

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 runtime 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.

If you drop and create indexes during periods of low system usage, you can reconfigure caches and pools and change cache bindings to optimize the sorts and reduce the time required. If you need to perform index maintenance while users are active, you need to consider the impact that 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 for **create index** take place in the cache to which the table is bound. If the table is not bound to a cache, but the database is, then cache is used. If there is no explicit cache binding, the default data cache is used. Worktable sorts use the cache to which *tempdb* is bound, or the default data cache.

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, use **sp_cacheconfig** check the space in the 2K and 16K pools in the cache.

How the Number of Sort Buffers Affects Sort Performance

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. During the sort, producers pass data to consumers using the sort buffers. This avoids disk I/O by copying data rows completely in memory. The reserved buffers are not available to any other tasks for the duration of the sort.

The configuration parameter **number of sort buffers** determines the maximum space that can be used to perform a serial sort. 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 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 “Sizing tempdb” on page 35-4 for more information.

- If you are creating indexes on large tables while other users are active, 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.
- If **number of sort buffers** is set to less than the square root of the worktable size, sort performance is degraded. Since worktables include only columns specified in the select list plus columns needed for later joins, worktable size for merge joins is usually considerably smaller than the original table size.

When enough 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 is 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 13-2.

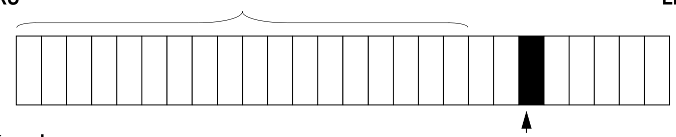


Figure 13-2: Area available for sort buffers

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: 38.23 Mb
      Config Replacement: strict LRU,   Run Replacement: strict LRU
      Config Partition:            2,   Run Partition:            2
 IO Size  Wash Size Config Size  Run Size     APF Percent
 -------- --------- ------------ ------------ -----------
     2 Kb   4544 Kb      0.00 Mb     22.23 Mb     10
    16 Kb   3200 Kb     16.00 Mb     16.00 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 “22.23 Mb” for the pool size and “4544 Kb” for the wash size:

```
bufs 22.23, 4544
```

The **bufs** procedure returns the following results:

```
90% of non-wash 2k pool
-----------------------
              8198.784
```

This command sets the number of sort buffers to 8198 pages:

```
sp_configure "number of sort buffers", 8198
```

If the table on which you want to create the index is bound to a user-defined cache, 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 Levels 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
as
declare @runs int, @merges int, @maxmerge int

select @runs = ceiling ( @pages / @bufs )

/* if all pages fit into sort buffers, no merge runs needed */
if @runs <=1
        select @merges = 0
else
```

```
begin
     if @runs > @bufs select @maxmerge = @bufs
     else  select @maxmerge = @runs

     if @maxmerge < 2 select @maxmerge = 2

     select @merges = ceiling(log10(@runs) / log10(@maxmerge))
end
select @merges "Merge Levels",
        2 * @pages * @merges + @pages "Total IO"
```

The parameters 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.

This example uses the default number of sort buffers for a table with 2,000,000 pages:

```
merge_runs 2000000, 500, 20
```

The **merge_runs** procedure estimates that 2 merge runs and 10,000,000 I/Os would be required to create the index:

```
Merge Levels Total IO
------------ -----------
           2    10000000
```

Increasing the number of sort buffers to 1500 reduces the number of merge runs and the I/O required:

```
merge_runs 2000000, 1500
```

```
Merge Levels Total IO
------------ -----------
           1     6000000
```

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, substantially reducing the number of physical I/Os for a sort. 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 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 helped performance slightly, since the small pool size tended to allow buffers to reach the LRU end of the cache before the writes were 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. See "Using sp_spaceused to Display Object Size" on page 15-4.
- **sp_estspace** – To predict the size of the index. See "Using sp_estspace to Estimate Object Size" on page 15-6.
- **sp_helpsegment** – To see space left on a database segment. See "Checking Data Distribution on Devices with sp_helpsegment" on page 33-29.

Space Requirements for Worktable Sorts

Queries that sort worktables (merge joins and **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 13-6, the number of devices in the target segment determines the number of consumers for 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 redoing 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 13-20.
- 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 11-12.
  - **sp_helppartition** – Displays information about partitioned tables. See "Getting Information About Partitions" on page 33-26.
  - **sp_helpsegment** – Displays information about segments, devices, and space usage. See "Checking Data Distribution on Devices with sp_helpsegment" on page 33-29.
  - **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 13-24.

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 13-2 describes the messages that can be printed for sort operations.

Table 13-2: Basic sort resource messages

Message	Explanation	See
`The Create Index is done using` *`sort_type`*	*sort_type* is either "Parallel Sort" or "Serial Sort."	"Parallel Sort Requirements and Resources Overview" on page 13-2
`Sort buffer size:` *`N`*	*N* is the configured value for the **number of sort buffers** configuration parameter.	"Sort Buffer Configuration Guidelines" on page 13-12
`Parallel degree:` *`N`*	*N* 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 13-11
`Number of output devices:` *`N`*	*N* is the total number of database devices on the target segment.	"Disk Requirements" on page 13-17
`Number of producer threads:` *`N`*	*N* is the optimal number of producer processes determined by the sort manager.	"Worker Process Requirements During Parallel Sorts" on page 13-6
`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 13-6
`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.	"Creating a Distribution Map" on page 13-4

Table 13-2: Basic sort resource messages (continued)

Message	Explanation	See
`Partition Element: N value`	*N* is the number of the distribution map element. *value* is the distribution map element that defines the boundary of each partition.	"Creating a Distribution Map" on page 13-4
`Number of sampled records: N`	*N* is the number of sampled records used to create the distribution map.	"Creating a Distribution Map" on page 13-4

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 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 worker processes. 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 number of worker processes used 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”

Query Tuning Tools

14 Introduction to Query Tuning Tools

This chapter provides a guide to the tools that can help you tune your queries. It contains the following sections:

The tools mentioned in this chapter are described in more detail in the chapters that follow.

Query Tuning Tools Overview

Adaptive Server provides the following diagnostic and informational tools to help you understand query optimization and improve the performance of your queries:

- A choice of tools to check or estimate the size of tables and indexes. These tools are described in Chapter 15, "Determining or Estimating the Sizes of Tables and Indexes."
- **set statistics io on** displays the number of logical and physical reads and writes required for each table in a query. If resource limits are enabled, it also displays the total actual I/O cost. **set statistics io** is described in Chapter 16, "Using the set statistics Commands."
- **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. See Chapter 17, "Using set showplan."
- **set statistics subquerycache on** displays the number of cache hits and misses and the number of rows in the cache for each subquery. See "Subquery Results Caching" on page 7-31 for examples.
- **set statistics time on** displays the time it takes to parse and compile each command. See "Checking Compile and Execute Time" on page 16-2 for more information.
- **dbcc traceon (302)** and **dbcc traceon(310)** provide additional information about why particular plans were chosen and is often used when the optimizer chooses a plan that seems incorrect. See Chapter 18, "Tuning with dbcc traceon."

- The **optdiag** utility command displays statistics for tables, indexes, and columns. See Chapter 19, "Statistics Tables and Displaying Statistics with optdiag."
- Chapter 20, "Advanced Optimizing Tools," explains tools you can use to enforce index choice, join order, and other query optimization choices. These tools include:
  - **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 with **(index...prefetch...mru_lru...parallel)** – Specifies the index, I/O size, or cache strategy to use for the query.
  - **set prefetch** –Toggles **prefetch** for query tuning experimentation.
  - **set sort_merge** – Disallows sort-merge joins.
  - **set parallel_degree** – Specifies the degree of parallelism for a query.
  - **sp_cachestrategy** – Sets status bits to enable or disable prefetch and fetch-and-discard cache strategies.

How Tools May Interact

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 when diagnostic information is being printed. 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. You can do so by using "echo input" flag to **isql**. 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 SQL statements from being executed.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.

How Tools Relate to Query Processing

Figure 14-1 shows how the query processing tools relate to different steps in query processing.

Many of the tools, for example, the **set** commands, affect the decisions made by the optimizer. **showplan** and **dbcc traceon(302, 310)** show you optimizer decision-making. **dbcc traceon(302,310)** shows intermediate information as analysis is performed, with **dbcc traceon(310)** printing the final plan statistics. **showplan** shows the final decision on access methods and join order.

statistics io and **statistics time** provide information about how the query was executed: **statistics time** measures time from the parse step until the query completes. **statistics io** prints actual I/O performed during query execution.

noexec allows you to obtain information such as **showplan** or **dbcc traceon(302,310)** output without actually executing the query.

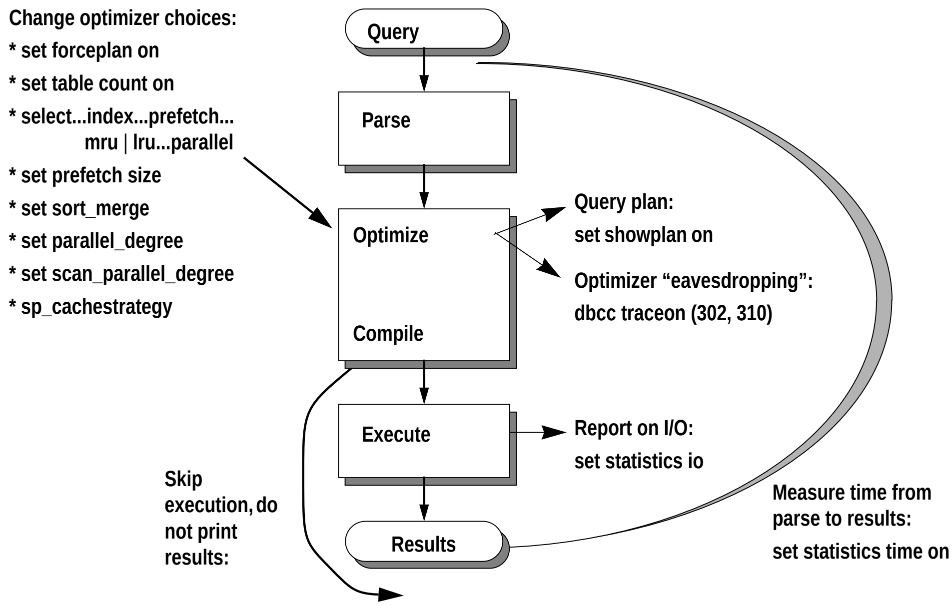


Figure 14-1: Query processing analysis tools and query processing

15 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 table size for space planning. It contains the following sections:

Why Object Sizes Are Important to Query Tuning

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

- Understand **statistics io** reports for a specific query plan. Chapter 16, “Using the set statistics Commands,” describes how to use **statistics io** to examine the I/O performed.
- Understand the optimizer’s choice of query plan. Adaptive Server’s cost-based optimizer estimates the physical and logical I/O required for each possible access method and chooses the cheapest method. If you think a particular query plan is unusual, you can used **dbcc traceon(302)** to determine why the optimizer made the decision. This output includes page number estimates.
- Determine object placement, based on the sizes of database objects and the expected I/O patterns on the objects. You can improve performance by distributing database objects across physical devices so that reads and writes to disk are evenly distributed. Object placement is described in Chapter 33, “Controlling Physical Data Placement.”
- Understand changes in performance. If objects grow, their performance characteristics can change. One example is a table that is heavily used and is usually 100 percent cached. If that table grows too large for its cache, queries that access the table can

suddenly suffer poor performance. This is particularly true for joins requiring multiple scans.

- Do capacity planning. Whether you are designing a new system or planning for growth of an existing system, you need to know the space requirements in order to plan for physical disks and memory needs.
- Understand output from Adaptive Server Monitor and from **sp_sysmon** reports on physical I/O.

Tools for Determining the Sizes of Tables and Indexes

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

- The utility program **optdiag** displays the sizes and many other statistics for tables and indexes. For information on using optdiag, see Chapter 19, "Statistics Tables and Displaying Statistics with optdiag."
- The system procedure **sp_spaceused** reports on the current size of an existing table and any indexes.
- The system procedure **sp_estspace** can predict the size of a table and its indexes, given a number of rows as a parameter.

You can also compute table and index size using formulas provided in this chapter. The **sp_spaceused** and **optdiag** commands report actual space usage. The other methods presented in this chapter provide size estimates. For partitioned tables, the system procedure **sp_helpartition** reports on the number of pages stored on each partition of the table. See Chapter 33, "Getting Information About Partitions," for information.

Effects of Data Modifications on Object Sizes

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

- When you insert a row that needs to be placed on a page of an allpages-locked table with a clustered index, and there is no room on the page for that row, the page is split, leaving two pages that are about 50 percent full.

- When you delete rows from heaps or from tables with clustered indexes, the space used on the page decreases. You can have pages that contain very few rows or even a single row.
- After some deletes or page splits have occurred, inserting rows into tables with clustered indexes tends to fill up pages that have been split or pages where rows have been deleted.

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

Using *optdiag* to Display Object Sizes

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

```
Table owner:                        "dbo"

Statistics for table:               "titles"

     Data page count:               662
     Empty data page count:         10
     Data row count:                4986.0000000000000000
     Forwarded row count:           18.0000000000000000
     Deleted row count:             87.0000000000000000
     Data page CR count:            86.0000000000000000
     OAM + allocation page count:   5
     First extent data pages:       3
     Data row size:                 238.8634175691937287
```

See Chapter 19, “Statistics Tables and Displaying Statistics with optdiag,” for more information.

Advantages of *optdiag*

The advantages of **optdiag** are:

- **optdiag** can display statistics for all tables in a database, or for a single table.
- **optdiag** output contains addition information useful for understanding query costs, such as index height and the average row length.

- **optdiag** is frequently used for other tuning tasks, so you should have these reports on hand.

Disadvantages of *optdiag*

The disadvantages of **optdiag** are:

- It produces a lot of output, so if you need only a single piece of information, such as the number of pages in the table, other methods are faster and have lower system overhead.

Using *sp_spaceused* to Display Object Size

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

```
sp_spaceused titles
```

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

The *rowtotal* value may be inaccurate at times; not all Adaptive Server processes update this value on the OAM page. The commands **update statistics**, **dbcc checktable**, and **dbcc checkdb** correct the *rowtotal* value on the OAM page. Table 15-1 explains the headings in **sp_spaceused** output.

Table 15-1: sp_spaceused output

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

To report index sizes separately, use:

```
sp_spaceused titles, 1
```

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

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

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

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

Advantages of *sp_spaceused*

The advantages of **sp_spaceused** are:

- It provides quick reports without excessive I/O and locking, since it uses only values in the table and index OAM pages to return results.
- It shows the amount of space that is reserved for expansion of the object, but not currently used to store data.
- It provides detailed reports on the size of indexes and of *text* and *image*, and Java off-row column storage.

Disadvantages of *sp_spaceused*

The disadvantages of **sp_spaceused** are:

- It may report inaccurate counts for row total and space usage.
- Output is in kilobytes, while most query-tuning activities use pages as a unit of measure.

Using *sp_estspace* to Estimate Object Size

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

To use **sp_estspace:**

- Create the table, if it does not exist.
- Create any indexes on the table.
- Execute the procedure, estimating the number of rows that the table will hold.

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

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

```
sp_estspace titles, 500000
```

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

```
Total_Mbytes
-----------------
  138.30

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

sp_estspace also allows you to specify a fillfactor, the average size of variable-length fields and text fields, and the I/O speed. For more information, see **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 perform initial capacity planning and to plan for table and index growth.
- **sp_estspace** helps you estimate the number of index levels.
- **sp_estspace** can be used to estimate future disk space, cache space, and memory requirements.

Disadvantages of *sp_estspace*

The disadvantages of using **sp_estspace** to estimate the sizes of objects are:

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

Using Formulas to Estimate Object Size

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

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

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

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

Factors That Can Affect Storage Size

Using space management properties can increase the space needed for a table or an index. See “Effects of Space Management Properties” on page 15-24, and “max_rows_per_page” on page 15-26.

The formulas in this section use the maximum size for variable-length character and binary data.To use the average size instead of the maximum size, see “Using Average Sizes for Variable Fields” on page 15-26.

If your table includes *text* or *image* datatypes or Java off-row columns, use 16 (the size of the text pointer that is stored in the row) in your calculations. Then see “LOB Pages” on page 15-26 to see how to calculate the storage space required for the actual *text* or *image* data.

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

- Duplicate keys are stored only once, followed by a list of row IDs for the key.
- Compression of keys on non-leaf levels; only enough of the key to differentiate from the neighboring keys is stored. This is

especially effective in reducing the size when long character keys are used.

If the configuration parameter **page utilization percent** is set to less than 100, Adaptive Server may allocate new extents before filling all pages on the allocated extents. This does not change the number of pages used by an object, but leaves empty pages in the extents allocated to the object. See “page utilization percent” on page 17-42 in the *System Administration Guide*.

Storage Sizes for Datatypes

The storage sizes for datatypes are shown in Table 15-2:

Table 15-2: Storage sizes for Adaptive Server datatypes

Datatype	Size
char	Defined size
nchar	Defined size * *@@ncharsize*
varchar	Actual number of characters
nvarchar	Actual number of characters * *@@ncharsize*
binary	Defined size
varbinary	Data size
int	4
smallint	2
tinyint	1
float	4 or 8, depending on precision
double precision	8
real	4
numeric	2–17, depending on precision and scale
decimal	2–17, depending on precision and scale
money	8
smallmoney	4
datetime	8

Table 15-2: Storage sizes for Adaptive Server datatypes (continued)

Datatype	Size
smalldatetime	4
bit	1
text	16 bytes + 2K * number of pages used
image	16 bytes + 2K * number of pages used
timestamp	8

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

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

All calculations in the examples that follow are based on the maximum size for *varchar*, *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 15-26.

Tables and Indexes Used in the Formulas

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

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

The table has two indexes:

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

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

- See “Calculating Table and Clustered Index Sizes for Allpages-Locked Tables” on page 15-11 for tables that use allpages-locking.
- See “Calculating the Sizes of Data-Only-Locked Tables” on page 15-18 for the formulas to use if tables that use data-only locking.

Calculating Table and Clustered Index Sizes for Allpages-Locked Tables

The formulas and examples for allpages-locked tables are divided into two sets of steps:

- Steps 1–6 outline the calculations for an allpages-locked table with a clustered index, giving the table size and the size of the index tree.
- Steps 7–12 outline the calculations for computing the space required by nonclustered indexes.

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

Step 1: Calculate the Data Row Size

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

Fixed-Length Columns Only

Use this formula if the table contains only fixed-length columns, and all are defined as NOT NULL.

Formula

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

Some Variable-Length Columns

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

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

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

Step 2: Compute the Number of Data Pages

Formula

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

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

Example

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

Step 3: Compute the Size of Clustered Index Rows

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

Fixed-Length Columns Only

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

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

Some Variable-Length Columns

	5	(Overhead)
+		Sum of bytes in the fixed-length index keys
+		Sum of bytes in variable-length index keys
		= Subtotal
+		(Subtotal / 256) + 1 (Overhead)
+		Number of variable-length columns + 1
+	2	(Overhead)
		= Clustered index row size

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

Step 4: Compute the Number of Clustered Index Pages

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

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

Formula

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

Example

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

Step 5: Compute the Total Number of Index Pages

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

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

Step 6: Calculate Allocation Overhead and Total Pages

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

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

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

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

Total Pages Needed

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

Formula			Example	
	Minimum	Maximum	Minimum	Maximum
Clustered index pages			3396	3379
OAM pages	+	+	1	2
Data pages	+	+	750000	750000
OAM pages	+	+	12	376
Total			753409	753773

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.

Fixed-Length Keys Only

Use this formula if the index contains only fixed-length keys and are defined as NOT NULL:

Formula

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

Some Variable-Length Keys

Use this formula if the index contains any variable-length keys or columns defined as NULL:

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

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

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

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

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

Step 10: Calculate the Number of Non-Leaf Pages

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

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

Formula

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

Example

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

Step 11: Calculate the Total Number of Non-Leaf Index Pages

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

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

Step 12: Calculate Allocation Overhead and Total Pages

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

Total Pages Needed

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

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

Calculating the Sizes of Data-Only-Locked Tables

The formulas and examples that follow show how to calculate the sizes of tables and indexes. This example uses the same columns sizes and index as the previous example. See “Tables and Indexes Used in the Formulas” on page 15-10 for the specifications.

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

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

Step 1: Calculate the Data Row Size

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

Fixed-Length Columns Only

Use this formula if the table contains only fixed-length columns defined as NOT NULL:

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

➤ ***Note***

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

Some Variable-Length Columns

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

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

Step 2: Compute the Number of Data Pages

Formula

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

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

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

Example

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

Step 3: Calculate Allocation Overhead and Total Pages

Allocation Overhead

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

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

Total Pages Needed

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

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

Step 4: Calculate the Size of the Index Row

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

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

Fixed-Length Keys Only

Use this formula if the index contains only fixed-length keys defined as NOT NULL:

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

Some Variable-Length Keys

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

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

Step 5: Calculate the Number of Leaf Pages in the Index

Formula

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

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

Example

2002 / 35 = 57 Nonclustered index rows per page

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

Step 6: Calculate the Number of Non-Leaf Pages in the Index

Formula

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

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

Formula

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

Example

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

Step 7: Calculate the Total Number of Non-Leaf Index Pages

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

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

Step 8: Calculate Allocation Overhead and Total Pages

Formula

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

Number of index pages / 2000 = Maximum OAM pages

Example

160713 / 63,750 = 3 (minimum)

160713 / 2000 = 81 (maximum)

Total Pages Needed

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

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

Other Factors Affecting Object Size

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

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

Effects of Space Management Properties

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

fillfactor

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

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

Other values for **fillfactor** reduce the number of rows per page on data pages and leaf index pages. To compute the correct values when using **fillfactor**, multiply the size of the available data page (2016) by the **fillfactor**. For example, if your **fillfactor** is 75 percent, your data page would hold 1471 bytes. Use this value in place of 2016 when you calculate the number of rows per page. For these calculations, see “Step 2: Compute the Number of Data Pages” on page 15-12 and “Step 8: Calculate the Number of Leaf Pages in the Index” on page 15-17.

exp_row_size

Setting an expected row size for a table can increase the amount of storage required. If your tables have many rows that are shorter than the expected row size, setting this value and running **reorg rebuild** or changing the locking scheme increases the storage space required for the table. However, the space usage for tables that formerly used **max_rows_per_page** should remain approximately the same. For more information, see “Reducing Row Forwarding with Expected Row Size” on page 31-7.

reservepagegap

Setting a **reservepagegap** for a table or an index leaves empty pages on extents that are allocated to the object when commands that perform extent allocation are executed. Setting **reservepagegap** to a low value increases the number of empty pages and spreads the data across more extents, so the additional space required is greatest immediately after a command such as **create index** or **reorg rebuild**. Row forwarding and inserts into the table fill in the reserved pages. For more information, see “Leaving Space for Forwarded Rows and Inserts” on page 31-13.

max_rows_per_page

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

To compute the correct values when using **max_rows_per_page**, use the **max_rows_per_page** value or the computed number of data rows per page, whichever is smaller, in“Step 2: Compute the Number of Data Pages” on page 15-12 and “Step 8: Calculate the Number of Leaf Pages in the Index” on page 15-17.

Using Average Sizes for Variable Fields

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

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

Very Small Rows

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

Number of Rows / 256 =	Number of data pages required

LOB Pages

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

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

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

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

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

```
Data length / 1800 = Number of 2K pages
```

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

The total space required for the data may be slightly larger than the calculated value, because some LOB pages store pointer information for other page chains in the column. Adaptive Server uses this pointer information to perform random access and prefetch data when accessing LOB columns. The additional space required to store pointer information depends on the total size and type of the data stored in the column. Use the information in Table 15-3 to estimate the additional pages required to store pointer information for data in LOB columns.

Table 15-3: Estimated additional pages for pointer information in LOB columns

Data Size and Type	Additional Pages Required for Pointer Information
400K *image*	0 to 1 page
700K *image*	0 to 2 pages
5MB *image*	1 to 11 pages
400K of multibyte *text*	1 to 2 pages
700K of multibyte *text*	1 to 3 pages
5MB of multibyte *text*	2 to 22 pages

Advantages of Using Formulas to Estimate Object Size

The advantages of using the formulas are:

- You learn more details of the internals of data and index storage.

- The formulas provide flexibility for specifying averages sizes for character or binary columns.
- While computing the index size, you see how many levels each index has, which helps estimate performance.

Disadvantages of Using Formulas to Estimate Object Size

The disadvantages of using the formulas are:

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

16 Using the *set statistics* Commands

This chapter contains the following sections:

set statistics Command Syntax

The syntax for the **set statistics** commands is:

```
set statistics {io, simulate, subquerycache, time}
   [on | off]
```

You can issue a single command:

```
set statistics io on
```

You can combine more than one command on a single line by separating them with commas:

```
set statistics io, time on
```

Using Simulated Statistics

The **optdiag** utility command allows you to load simulated statistics and perform query diagnosis using those statistics. Since you can load simulated statistics even for tables that are empty, using simulated statistics allows you to perform tuning diagnostics in a very small database that contains only the tables and indexes. Simulated statistics do not overwrite any existing statistics when they are loaded, so you can also load them into an existing database.

Once simulated statistics have been loaded, instruct the optimizer to use them (rather than the actual statistics):

```
set statistics simulate on
```

For complete information on using simulated statistics, see "Using Simulated Statistics" on page 19-26.

Checking Subquery Cache Performance

When subqueries are not flattened or materialized, a subquery cache is created to store results of earlier executions of the subquery to reduce the number of expensive executions of the subquery. See "Displaying Subquery Cache Information" on page 7-32 for information on using this option.

Checking Compile and Execute Time

set statistics time displays information about the time it takes to parse and execute Adaptive Server commands.

```
Parse and Compile Time 57.
SQL Server cpu time: 5700 ms.

Execution Time 175.
SQL Server cpu time: 17500 ms.  SQL Server elapsed
time: 70973 ms.
```

The meaning of this output is:

- Parse and Compile Time – The number of CPU ticks taken to parse, optimize, and compile the query. See below for information on converting ticks to milliseconds.
- SQL Server cpu time – Shows the CPU time in milliseconds.
- Execution Time – The number of CPU ticks taken to execute the query.
- SQL Server cpu time – The number of CPU ticks taken to execute the query, converted to milliseconds.
- SQL Server elapsed time – The difference in milliseconds between the time the command started and the current time, as taken from the operating system clock.

This output shows that the query was parsed and compiled in 57 clock ticks. It took 175 ticks, or 17.5 seconds, of CPU time to execute. Total elapsed time was 70.973 seconds, indicating that Adaptive Server spent some time processing other tasks or waiting for disk or network I/O to complete.

Converting Ticks to Milliseconds

To convert ticks to milliseconds:

$$\text{Milliseconds} = \frac{CPU\_ticks * clock\_rate}{1000}$$

To see the *clock_rate* for your system, execute:

```
sp_configure "sql server clock tick length"
```

See “sql server clock tick length” on page 17-148 of the *System Administration Guide* for more information.

Reporting Physical and Logical I/O Statistics

set statistics io reports information about physical and logical I/O and the number of times a table was accessed. **set statistics io** output follows the query results and provides 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:

Cost = All physical IOs * 18 + All logical IOs * 2

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 the example above that performs 656 physical reads and 656 logical reads, 656 * 2 + 656 * 18 = 13120, which is the total I/O cost reported by **statistics io**.

Statistics for Writes

statistics io reports the total number of buffers written by the command. Read-only queries report writes when they cause dirty pages to move past the wash marker in the cache so that the write on the 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

statistics io reports the number of logical and physical reads for each table and index included in a query, including worktables. I/O for indexes is included with the I/O for the table.

Figure 16-1 shows the values that **statistics io** reports for logical and physical reads.

Table 16-1: statistics io output for reads

Output	Description
logical reads	
regular	Number of times that a page needed by the query was found in cache; only pages not brought in by asynchronous prefetch (APF) are counted here.
apf	Number of times that a request brought in by an APF request was found in cache.
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 w.as 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.
- All of the pages read by APF were used by the query.

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.

statistics io Output for Cursors

For queries using cursors, **statistics io** prints the cumulative I/O since the cursor was opened:

```
1> open c

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
Total actual I/O cost for this command: 0.
Total writes for this command: 0

1> fetch c

title_id type         price
 -------- ------------ -----------------------
 T24140   business                      201.95
Table: titles scan count 1, logical reads:
(regular=3 apf=0 total=3), physical reads:
(regular=0 apf=0 total=0), apf IOs used=0
Total actual I/O cost for this command: 6.
Total writes for this command: 0

1> fetch c
```

```
title_id type         price
 -------- ------------ ------------------------
 T24226   business                        201.95
Table: titles scan count 1, logical reads:
(regular=4 apf=0 total=4), physical reads:
(regular=0 apf=0 total=0), apf IOs used=0
Total actual I/O cost for this command: 8.
Total writes for this command: 0
```

Scan Count

statistics io reports the number of times a query accessed a particular table. 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 as a scan.
- 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 access to the table is counted as a scan.

Use **showplan**, as described in Chapter 17, “Using set showplan,” 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 scan the tables; otherwise, they perform table scans. In either case, they require only a single scan 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 that report a scan for each worker process.
- Queries that have indexed **where** clauses connected by **or** report a scan for each **or** clause. If the query uses the special OR strategy, it reports one scan for each value. If the query uses the OR strategy, it reports one scan for each index, plus one scan for the RID list access.

  This query uses the special OR strategy, so it reports a scan count of 2 if the *titles* table has indexes on *title_id* and another on *pub_id*:

```
select title_id
from titles
    where title_id = "T55522"
    or pub_id  = "P988"
```

```
Table: titles scan count 2,logical reads: (regular=149 apf=0 total=149),
physical reads: (regular=63 apf=80 total=143), apf IOs used=80

Table: Worktable1  scan count 1, logical reads: (regular=172 apf=0
total=172), physical reads: (regular=0 apf=0 total=0), apf IOs
```

  The I/O for the worktable is also reported.

- Nested-loop joins that scan inner tables 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=442 apf=0 total=442),
physical reads: (regular=53 apf=289 total=342), apf IOs used=289
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 a table scan on *publishers*, which 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, using an index on *pub_id*.

- Merge joins with duplicate values in the outer table restart the scan for each duplicate value, and report an additional scan count each time.

Queries Reporting Scan Count of 0

Multistep queries and certain other types of queries may report a scan count of 0. Some examples are:

- Queries that perform deferred updates
- **select...into** queries
- Queries that create worktables

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.

When pages for the query are found in cache, logical reads are higher than physical reads. This happens frequently with pages from higher levels of the index, since they are reused often, and tend to remain in cache.

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 (**showplan** reports the I/O size), a single physical read brings 8 data pages into cache. If a query reports 100 16K physical reads, it has read 800 data pages into cache. 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 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.

During actual use on a production system, a query can be expected to find some of the required pages already in the cache, from earlier access by other users, while other pages need to be read from disk. Higher levels of indexes, in particular, tend to be frequently used, and tend to remain in the cache. If you have a table or index bound to a cache that is large enough to hold all the pages, no physical I/O takes place once the object has been read into cache.

However, during query tuning on a development system with few users, you may want to clear the pages used for the query from cache in order to see the full physical I/O needed for a query. You can clear an object's pages from cache by:

- Changing the cache binding for the object:
  - If a table or index is bound to a cache, unbind it, and rebind it.
  - If a table or index is not bound to a cache, bind it to any cache available, then unbind it.

  You must have at least one user-defined cache to use this option.
- If you do not have any user-defined caches, you can execute a sufficient number of queries on other tables, so that the objects of interest are flushed from cache. If the cache is very large, this can be time-consuming.
- The only other alternative is rebooting the server.

For more information on testing and cache performance, see "Testing Data Cache Performance" on page 32-10.

statistics io and Merge Joins

statistics io output does not include sort costs for merge joins. If you have **allow resource limits** enabled, the sort cost is not reported in the "Total estimated I/O cost" and "Total actual I/O cost" statistics. Only **dbcc traceon(310)** shows these costs.

17 Using *set showplan*

This chapter describes each message printed by the **showplan** utility. **showplan** 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*

To see the query plan for a query, use:

```
set showplan on
```

To stop displaying query plans, use:

```
set showplan off
```

You can use **showplan** in conjunction with other **set** commands. See Chapter 14, "Introduction to Query Tuning Tools," for information on how options affect each other's operation.

Basic *showplan* Messages

This section describes **showplan** messages that are printed for most **select**, **insert**, **update**, and **delete** operations.

Table 17-1: Basic showplan messages

Message	Explanation	See
`QUERY PLAN FOR STATEMENT N (at line N).`	"Statement *N*" is the statement number within the batch; "line *N*" is the line number within the batch.	Page 17-2

Table 17-1: Basic showplan messages (continued)

Message	Explanation	See
`STEP N`	Each step for each statement is numbered sequentially. Numbers are restarted at 1 on each side of a **union.**	Page 17-3
`The type of query is` *`query type`*`.`	*query type* is replaced by the type of query: SELECT, UPDATE, INSERT, or any Transact-SQL statement type.	Page 17-3
`FROM TABLE`	Each occurrence of FROM TABLE indicates a table to be read. The table name is listed on the next line. Table 17-3 on page 17-22 shows the access method messages that **showplan** displays for each table access.	Page 17-4
`TO TABLE`	Included when a command creates a worktable and for **insert...select** commands.	Page 17-6

`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 **insert**, **delete**, or **update** is performed in direct or deferred update mode. See "Update Mode Messages" on page 17-7.	Page 17-8

`Optimized using simulated statistics.`
`Optimized using an Abstract Plan (ID : N).`
`Optimized using the Abstract Plan in the PLAN clause.`
	The query was either optimized using simulated statistics, or using an abstract plan.	Page 17-10

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 section of **showplan** output from the next section. 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 retrieve the results in a single step and breaks 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.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.
```

Multiple-step queries are demonstrated following "GROUP BY Message" on page 17-12.

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:

```
STEP 1
        The type of query is CREATE INDEX.
        TO TABLE
            titleauthor
```

FROM TABLE Message

```
FROM TABLE
    tablename [ correlation_name ]
```

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 names. 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 plan chosen by the optimizer joins the tables. 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
        Forward scan.
        Positioning by key.
        Keys are:
            au_lname  ASC
        Using I/O Size 2 Kbytes for index leaf pages.
        With LRU Buffer Replacement Strategy for index leaf pages.
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.

        FROM TABLE
            titleauthor
            ta
        Nested iteration.
```

```
Index : at_ix
Forward scan.
Positioning by key.
Index contains all needed columns. Base table will not be read.
Keys are:
    au_id  ASC
Using I/O Size 2 Kbytes for index leaf pages.
With LRU Buffer Replacement Strategy for index leaf pages.

FROM TABLE
    titles
    t
Nested iteration.
Using Clustered Index.
Index : title_id_ix
Forward scan.
Positioning by key.
Index contains all needed columns. Base table will not be read.
Keys are:
    title_id  ASC
Using I/O Size 2 Kbytes for index leaf pages.
With LRU Buffer Replacement Strategy for index leaf pages.
```

The sequence of tables in this output shows the order chosen by the 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. 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
        Forward scan.
        Positioning by key.
        Keys are:
            title_id
        Using I/O Size 2 Kbytes for index leaf pages.
        With LRU Buffer Replacement Strategy for index leaf pages.
        TO TABLE
            salesdetail
```

The clustered index on *title_id_ix* is used to verify 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. This **insert** command shows 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
```

Here is a command that performs an update:

```
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.
       Using Clustered Index.
       Index : publ_id_ix
       Forward scan.
       Positioning by key.
       Keys are:
           pub_id  ASC
       Using I/O Size 2 Kbytes for index leaf pages.
       With LRU Buffer Replacement Strategy for index leaf pages.
       Using I/O Size 2 Kbytes for data pages.
       With LRU Buffer Replacement Strategy for data pages.
       TO TABLE
           publishers
```

The update query output 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.

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.

For more information on the different types of direct updates, see “How Update Operations Are Performed” on page 6-30.

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.
        Using Clustered Index.
        Index : au_names_ix
        Forward scan.
        Positioning by key.
        Keys are:
            au_lname  ASC
            au_fname  ASC
        Using I/O Size 2 Kbytes for index leaf pages.
        With LRU Buffer Replacement Strategy for index leaf pages.
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.
        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.

See “How Update Operations Are Performed” on page 6-30.)

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.
     Forward scan.
     Positioning at start of table.
     Using I/O Size 2 Kbytes for data pages.
     With LRU Buffer Replacement Strategy for data pages.
     TO TABLE
         mytable
```

This command copies every row in the table and appends the rows to the end of the table. It 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 runtime.

Adaptive Server uses deferred_index mode when the index 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 6-34.

Optimized Using Messages

These messages are printed when special optimization options are used for a query.

Simulated Statistics Message

```
Optimized using simulated statistics.
```

The simulated statistics message is printed when:

- The **set statistics simulate** option was active when the query was optimized, and
- Simulated statistics have been loaded using **optdiag**.

See “Using Simulated Statistics” on page 19-26 for more information.

Abstract Plan Messages

```
Optimized using an Abstract Plan (ID : N).
```

The message above is printed when an abstract plan was associated with the query. The variable prints the ID number of the plan.

```
Optimized using the Abstract Plan in the PLAN
clause.
```

The message above is printed when the **plan** clause is used for a **select**, **update**, or **delete** statement. See Chapter 22, “Creating and Using Abstract Plans,” for more information.

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 17-2: showplan messages for various clauses

Message	Explanation	See
`GROUP BY`	The query contains a **group by** statement.	Page 17-12
`The type of query is SELECT (into WorktableN)`	The step creates a worktable to hold intermediate results.	Page 17-12

`Evaluate Grouped type AGGREGATE`
or
`Evaluate Ungrouped type 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 17-13
		Page 17-16

`Evaluate Grouped ASSIGNMENT OPERATOR` `Evaluate Ungrouped ASSIGNMENT OPERATOR`	Query includes **compute** (ungrouped) or **compute by** (grouped).	Page 17-14
`WorktableN created for DISTINCT.`	The query contains a **distinct** keyword in the select list that requires a sort to eliminate duplicates.	Page 17-18
`WorktableN created for ORDER BY.`	The query contains an **order by** clause that requires ordering rows.	Page 17-19
`This step involves sorting.`	The query includes on **order by** or **distinct** clause, and results must be sorted.	Page 17-20

Table 17-2: showplan messages for various clauses (continued)

Message	Explanation	See
`Using GETSORTED.`	The query created a worktable and sorted it. GETSORTED is a technique used to return the rows.	Page 17-21
`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 17-21

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.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 16 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.
        TO TABLE
            Worktable1.

    STEP 2
        The type of query is SELECT.

        FROM TABLE
            Worktable1.
        Nested iteration.
        Table Scan.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 16 Kbytes for data pages.
        With MRU Buffer Replacement Strategy for data pages.
```

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 17-58.

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.
       Forward scan.
       Positioning at start of table.
       Using I/O Size 16 Kbytes for data pages.
       With LRU Buffer Replacement Strategy for data pages.
       TO TABLE
           Worktable1.

   STEP 2
       The type of query is SELECT.

       FROM TABLE
           Worktable1.
       Nested iteration.
       Table Scan.
       Forward scan.
       Positioning at start of table.
       Using I/O Size 16 Kbytes for data pages.
       With MRU Buffer Replacement Strategy for data pages.
```

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
        Forward scan.
        Positioning by key.
        Keys are:
            title  ASC
        Using I/O Size 2 Kbytes for index leaf pages.
        With LRU Buffer Replacement Strategy for index leaf pages.
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.
        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.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 16 Kbytes for data pages.
        With MRU Buffer Replacement Strategy for data pages.
```

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 17-13 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_price
        Forward scan.
        Positioning by key.
        Keys are:
            type  ASC
        Using I/O Size 2 Kbytes for index leaf pages.
        With LRU Buffer Replacement Strategy for index leaf pages.
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.

    STEP 2
        The type of query is SELECT.
```

This a two-step query, 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
where title like "Compu%"
order by type
compute avg(advance)
```

The **showplan** output shows that the computation of the aggregate values takes place in the 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
        Forward scan.
        Positioning by key.
        Keys are:
            title  ASC
        Using I/O Size 2 Kbytes for index leaf pages.
        With LRU Buffer Replacement Strategy for index leaf pages.
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.
        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.
Forward scan.
Positioning at start of table.
Using I/O Size 16 Kbytes for data pages.
With MRU Buffer Replacement Strategy for data pages.
```

Messages for *order by* and *distinct*

Some queries that include **distinct** use a sort step to enforce the uniqueness of values in the result set. **distinct** queries and **order by** queries do not require the sorting step when the index used to locate rows supports the **order by** or **distinct** clause.

For those cases where the sort is 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, discards the duplicate rows, and then returns the rows.

Subqueries with existence joins sometimes create a worktable and sort it to remove duplicate rows. See "Flattening in, any, and exists Subqueries" on page 7-23 for more information.

The "Worktable*N* 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 17-20.

```
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.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 16 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.
        TO TABLE
            Worktable1.

    STEP 2
        The type of query is SELECT.
        This step involves sorting.

        FROM TABLE
            Worktable1.
        Using GETSORTED
        Table Scan.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 16 Kbytes for data pages.
        With MRU Buffer Replacement Strategy for data pages.
```

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.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 16 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.
        TO TABLE
            Worktable1.

    STEP 2
        The type of query is SELECT.
        This step involves sorting.

        FROM TABLE
            Worktable1.
        Using GETSORTED
        Table Scan.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 16 Kbytes for data pages.
        With MRU Buffer Replacement Strategy for data pages.
```

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 Chapter 9, "Indexing for Performance," 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 17-18 and "Worktable Message for order by" on page 17-19.

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 17-18 and for **order by** on page 17-19 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 after the sort manager 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 is not performed, 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 17-3: showplan messages describing access methods

Message	Explanation	See
`Auxiliary scan descriptors required: N`	*N* indicates the number of scan descriptors needed to run the query.	Page 17-23
`Nested iteration.`	Indicates the execution of a data retrieval loop.	Page 17-25
`Merge join (outer table).` `Merge join (inner table).`	Indicates that a merge join is being performed, and whether this table is the outer or inner table.	Page 17-25
`WorktableN created for sort merge join.`	Indicates that a worktable is being used for a merge join.	Page 17-28
`Table Scan.`	Indicates that the query performs a table scan.	Page 17-28
`Using Clustered Index.`	Query uses the clustered index on the table.	Page 17-29
`Index : index_name`	Query uses an index on the table; the variable shows the index name.	Page 17-29
`Forward scan.`	Indicates a table or index scan searching from the beginning of the table or index.	Page 17-30
`Backward scan.`	Indicates a table or index scan searching from the end of the table or index.	Page 17-30

`Positioning at start of table.`
`Positioning at end 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 17-31

`Scanning only up to the first qualifying row.` `Scanning only the last page of the table.`	These messages indicate **min** and **max** optimization, respectively.	Page 17-33

Table 17-3: showplan messages describing access methods (continued)

Message	Explanation	See
`Index contains all needed columns. Base table will not be read.`	Indicates that the nonclustered index covers the query.	Page 17-33
`Keys are:`	Included when the positioning message indicates "Positioning by key." The next line(s) shows the index key(s) used.	Page 17-34
`ASC` `DESC`	Indicates the order of keys in the index used for the query.	Page 17-34
`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 (values list)** item.	Page 17-35
`Using Dynamic Index.`	Reported during some queries using **or** clauses or **in (values list).**	Page 17-36
`Worktable`*`N`* `created for REFORMATTING.`	Indicates that the reformatting strategy is being used.	Page 17-37
`Log Scan.`	Query fired a trigger that uses *inserted* or *deleted* tables.	Page 17-39
`Using I/O Size` *`N`* `Kbytes for data pages.` `Using I/O size` *`N`* `Kbytes for index leaf pages.`	*N* indicates the I/O size for physical I/O for the data and index leaf pages.	Page 17-41
`With <LRU/MRU> buffer replacement strategy for data pages.` `With <LRU/MRU> Buffer Replacement Strategy for index leaf pages.`	Reports the caching strategy for the table or index.	Page 17-41
`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 17-41

Auxiliary Scan Descriptors Message

```
Auxiliary scan descriptors required: N
```

When a query involving referential integrity requires a large number of user or system tables, including references to other tables to check referential integrity, this **showplan** message indicates the number of auxiliary scan descriptors needed for the query. If a query does not exceed the number of preallocated scan descriptors allotted for the session, the "Auxiliary scan descriptors required" message is not printed.

The following example shows partial output for a delete from the *employees* table, which 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: 4


    STEP 1
        The type of query is DELETE.
        The update mode is direct.

        FROM TABLE
            employees
        Nested iteration.
        Using Clustered Index.
        Index : employees_empl_i_10080066222
        Forward scan.
        Positioning by key.
        Keys are:
            empl_id  ASC
        Using I/O Size 2 Kbytes for index leaf pages.
        With LRU Buffer Replacement Strategy for index leaf pages.
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.

        FROM TABLE
            benefits
        Index : empl_id_ix
        Forward scan.
        Positioning by key.
        Index contains all needed columns. Base table will not be read.
        Keys are:
            empl_id  ASC
        Using I/O Size 2 Kbytes for index leaf pages.
        With LRU Buffer Replacement Strategy for index leaf pages.
        .
        .
        .
        FROM TABLE
            dependents
```

```
        Index : empl_id_ix
        Forward scan.
        Positioning by key.
        Index contains all needed columns. Base table will not be read.
        Keys are:
            empl_id  ASC
        Using I/O Size 2 Kbytes for index leaf pages.
        With LRU Buffer Replacement Strategy for index leaf pages.
        TO TABLE
            employees
```

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, as shown here:

```
select * from publishers
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).

    STEP 1
        The type of query is SELECT.

        FROM TABLE
            publishers
        Nested iteration.
        Table Scan.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.
```

For queries that perform nested-loop joins, access to each table is nested within the scan of the outer table. See “Nested-Loop Joins” on page 7-6 for more information.

Merge Join Messages

```
Merge join (outer table).
```

```
Merge join (inner table).
```

Merge join messages indicate the use of a merge join and the table’s position (inner or outer) with respect to the other table in the merge join. Merge join messages appear immediately after the table name in the `FROM TABLE` output. This query performs a mixture of merge and nested-loop joins:

```
select pub_name, au_lname, price
from titles t, authors a, titleauthor ta,
            publishers p
where t.title_id = ta.title_id
    and a.au_id = ta.au_id
    and p.pub_id = t.pub_id
    and type = 'business'
    and price < $25
```

Messages for merge joins are printed in bold type in the **showplan** output:

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 3 worker processes.


    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
            titles
            t
        Merge join (outer table).
        Parallel data merge using 3 worker processes.
        Using Clustered Index.
        Index : title_id_ix
        Forward scan.
        Positioning by key.
        Keys are:
            title_id  ASC
        Using I/O Size 16 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.

        FROM TABLE
            titleauthor
            ta
        Merge join (inner table).
        Index : ta_ix
        Forward scan.
        Positioning by key.
        Index contains all needed columns. Base table will not be read.
        Keys are:
            title_id  ASC
        Using I/O Size 16 Kbytes for index leaf pages.
        With LRU Buffer Replacement Strategy for index leaf pages.

        FROM TABLE
            authors
            a
```

```
    Nested iteration.
    Index : au_id_ix
    Forward scan.
    Positioning by key.
    Keys are:
        au_id  ASC
    Using I/O Size 2 Kbytes for index leaf pages.
    With LRU Buffer Replacement Strategy for index leaf pages.
    Using I/O Size 2 Kbytes for data pages.
    With LRU Buffer Replacement Strategy for data pages.
    TO TABLE
        Worktable1.
    Worktable1 created for sort merge join.

STEP 2
    The type of query is INSERT.
    The update mode is direct.
    Executed by coordinating process.

    FROM TABLE
        publishers
        p
    Nested iteration.
    Table Scan.
    Forward scan.
    Positioning at start of table.
    Using I/O Size 2 Kbytes for data pages.
    With LRU Buffer Replacement Strategy for data pages.
    TO TABLE
        Worktable2.
    Worktable2 created for sort merge join.

STEP 3
    The type of query is SELECT.
    Executed by coordinating process.

    FROM TABLE
        Worktable1.
    Merge join (outer table).
    Serial data merge.
    Table Scan.
    Forward scan.
    Positioning at start of table.
    Using I/O Size 2 Kbytes for data pages.
    With LRU Buffer Replacement Strategy for data pages.

    FROM TABLE
        Worktable2.
    Merge join (inner table).
    Table Scan.
```

```
        Forward scan.
        Positioning at start of table.
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.

Total estimated I/O cost for statement 1 (at line 1): 4423.

The sort for Worktable1 is done in Serial

The sort for Worktable2 is done in Serial
```

This query performed the following joins:

- A full-merge join on *titles* and *titleauthor*, with *titles* as the outer table
- A nested-loop join with the *authors* table
- A sort-merge join with the *publishers* table

Worktable Message

```
WorktableN created for sort merge join.
```

If a merge join requires a sort for a table, a worktable is created and sorted into order by the join key. A later step in the query uses the worktable as either an inner table or outer table.

Table Scan Message

```
Table Scan.
```

This message indicates that the query performs a table scan. 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.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 16 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.
```

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
        Forward scan.
        Positioning by key.
        Keys are:
            title_id  ASC
        Using I/O Size 16 Kbytes for index leaf pages.
        With LRU Buffer Replacement Strategy for index leaf pages.
        Using I/O Size 16 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.
```

Index Name Message

```
Index : indexname
```

This message indicates that the query 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: key [ASC | DESC] ..." on page 17-34.

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_names_ix
        Forward scan.
        Positioning by key.
        Keys are:
            au_fname  ASC
        Using I/O Size 16 Kbytes for index leaf pages.
        With LRU Buffer Replacement Strategy for index leaf pages.
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.
```

Scan Direction Messages

```
Forward scan.
```

```
Backward scan.
```

These messages indicate the direction of a table or index scan.

The scan direction depends on the ordering specified when the indexes were created and the order specified for columns in the **order by** clause.

Backward scans cam be used when the **order by** clause contains the **asc** or **desc** qualifiers on index keys, in the exact opposite of those in the **create index** clause. The configuration parameter **allow backward scans** must be set to 1 to allow backward scans.

The scan-direction messages are followed by positioning messages. Any keys used in the query are followed by “ASC” or “DESC”. The forward and backward scan messages and positioning messages describe whether a scan is positioned:

- At the first matching index key, at the start of the table, or at the first page of the leaf-level pages chain, and searching toward end of the index, or
- At the last matching index key, or end of the table, or last page of the leaf-level page chain, and searching toward the beginning.

If **allow backward scans** is set to 0, all accesses use forward scans.

This example uses a backward scan:

```
select *
from sysmessages
where description like "%Optimized using%"
order by error desc
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).

    STEP 1
        The type of query is SELECT.

        FROM TABLE
            sysmessages
        Nested iteration.
        Table Scan.
        Backward scan.
        Positioning at end of table.
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.
```

This query using the **max** aggregate also uses a backward scan:

```
select max(error) from sysmessages
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).

    STEP 1
        The type of query is SELECT.
        Evaluate Ungrouped MAXIMUM AGGREGATE.

        FROM TABLE
            sysmessages
        Nested iteration.
        Index : ncsysmessages
        Backward scan.
        Positioning by key.
        Scanning only up to the first qualifying row.
        Index contains all needed columns. Base table will not be read.
        Keys are:
            error  ASC
        Using I/O Size 2 Kbytes for index leaf pages.
        With LRU Buffer Replacement Strategy for index leaf pages.

    STEP 2
        The type of query is SELECT.
```

Positioning Messages

```
Positioning at start of table.
Positioning at end 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 an index takes place. The choices are:

- `Positioning at start of table.`

  This message indicates a forward table scan, starting at the first row of the table.

- `Positioning at end of table.`

  This message indicates a backward table scan, starting at the last 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 (OR Strategy)" on page 17-36 for more information about how row IDs are used.

- `Positioning by key.`

  This messages indicates that the index is used to position the search at the first qualifying row. It is printed for:

  - Direct access an individual row in a point query
  - Range queries that perform matching scans of the leaf level of an index
  - Range queries that scan the data pages when there is a clustered index on an allpages-locked table
  - Indexed accesses to inner tables in joins

- `Positioning at index start.`
  `Positioning at index end.`

  These messages indicate a nonmatching index scan, used when the index covers the query. Matching scans are positioned by key.

  Forward scans are positioned at the start of the index; backward scans are positioned at the end of the index.

Scanning Messages

```
Scanning only the last page of the table.
```

This message indicates that a query containing an ungrouped (scalar) **max** aggregate can access only the last page of the table to return the value. See "How Aggregates Are Optimized" on page 6-28 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 "How Aggregates Are Optimized" on page 6-28 for more information.

➤ ***Note***

For indexes with the leading key created in descending order, the use of the messages for min and max aggregates is reversed: min uses "Positioning at index end" while max prints "Positioning at index start" and "Scanning only up to the first qualifying row."

Index Covering Message

```
Index contains all needed columns. Base table will
not be read.
```

This message indicates that an index covers the query. It is printed both for matching and nonmatching scans. Other messages in **showplan** output help distinguish these access methods:

- A matching scan reports "Positioning by key." A nonmatching scan reports "Positioning at index start," or "Positioning at index end" since a nonmatching scan must read the entire leaf level of the index.
- If the optimizer uses a matching scan, the "Keys are..." message reports the keys used to position the search. This message is not included for a nonmatching scan.

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
        Forward scan.
        Positioning by key.
        Index contains all needed columns. Base table will not be read.
        Keys are:
            au_lname  ASC
        Using I/O Size 2 Kbytes for index leaf pages.
        With LRU Buffer Replacement Strategy for index leaf pages.
```

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"
```

```
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
        Forward scan.
        Positioning at index start.
        Index contains all needed columns. Base table will not be read.
        Using I/O Size 16 Kbytes for index leaf pages.
        With LRU Buffer Replacement Strategy for index leaf pages.
```

Note that the **showplan** output does not contain a “Keys are...” message, and the positioning message is “Positioning at index start.” This query scans the entire leaf level of the nonclustered index, since the rows are not ordered by the search argument.

Keys Message

```
Keys are:
    key [ ASC | DESC ] ...
```

This message is followed by the index key(s) used when Adaptive Server uses an index scan to locate rows. The index ordering is printed after each index key, showing the order, ASC for ascending or DESC for descending, used when the index was created. For composite indexes, all leading keys in the **where** clauses are listed.

Matching Index Scans Message

```
Using N Matching Index Scans.
```

This **showplan** message indicates that a query using **or** clauses or an **in (*values list*)** clause uses multiple index scans (also called the "special OR strategy") instead of using a dynamic index.

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 more information on how queries containing **or** are processed, see "Access Methods and Costing for or and in Clauses" on page 6-23.

For queries that use multiple matching scans, 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
        Forward scan.
        Positioning by key.
        Keys are:
            title_id
```

```
Index : title_id_ix
Forward scan.
Positioning by key.
Keys are:
    title_id
Using I/O Size 2 Kbytes for data pages.
With LRU Buffer Replacement Strategy for data pages.
```

Dynamic Index Message (OR Strategy)

```
Using Dynamic Index.
```

The term **dynamic index** refers to a worktable of row IDs used to process some queries that use **or** clauses or an **in (values list)** clause. 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. For a full explanation, see “Access Methods and Costing for or and in Clauses” on page 6-23.

For a query with two SARGs that match the two indexes (one on *au_fname*, one on *au_lname*), 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 the first name “William” and one for the last name “Williams”. These blocks include the output “Index contains all needed columns,” since the row IDs can be retrieved from the leaf level of a nonclustered index.
- The final “FROM TABLE” block shows the “Using Dynamic Index” output and “Positioning by Row IDentifier (RID).” In this step, the dynamic index is used to access the data pages to locate the rows to be returned.

```
select au_id, au_fname, au_lname
from authors
where au_fname = "William"
    or 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_fname_ix
        Forward scan.
```

```
Positioning by key.
Index contains all needed columns. Base table will not be read.
Keys are:
    au_fname  ASC
Using I/O Size 2 Kbytes for index leaf pages.
With LRU Buffer Replacement Strategy for index leaf pages.

FROM TABLE
    authors
Nested iteration.
Index : au_lname_ix
Forward scan.
Positioning by key.
Index contains all needed columns. Base table will not be read.
Keys are:
    au_lname  ASC
Using I/O Size 2 Kbytes for index leaf pages.
With LRU Buffer Replacement Strategy for index leaf pages.

FROM TABLE
    authors
Nested iteration.
Using Dynamic Index.
Forward scan.
Positioning by Row IDentifier (RID).
Using I/O Size 2 Kbytes for data pages.
With LRU Buffer Replacement Strategy for data pages.
```

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 keys to join the worktable to the other table in the query.
- Uses the clustered index in the join to retrieve the qualifying rows from the table.

See “The Reformatting Strategy” on page 7-21 for more information on reformatting.

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*), so Adaptive Server uses 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.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.
        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.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.
        TO TABLE
            Worktable2.

    STEP 3
        The type of query is SELECT.

        FROM TABLE
```

```
    titles
    t
Nested iteration.
Table Scan.
Forward scan.
Positioning at start of table.
Using I/O Size 2 Kbytes for data pages.
With LRU Buffer Replacement Strategy for data pages.

FROM TABLE
    Worktable1.
Nested iteration.
Using Clustered Index.
Forward scan.
Positioning by key.
Using I/O Size 2 Kbytes for data pages.
With LRU Buffer Replacement Strategy for data pages.

FROM TABLE
    Worktable2.
Nested iteration.
Using Clustered Index.
Forward scan.
Positioning by key.
Using I/O Size 2 Kbytes for data pages.
With LRU Buffer Replacement Strategy for data pages.
```

This query was run with **set sort_merge off**. When sort-merge joins are enabled, this query chooses a sort-merge join instead.

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. This example shows the output for the update to the *titles* table when this insert fires the *totalsales_trig* trigger on the *salesdetail* table:

```
insert salesdetail values ('7896', '234518',
'TC3218', 75, 40)
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).

    STEP 1
        The type of query is UPDATE.
        The update mode is direct.

        FROM TABLE
            titles
        Nested iteration.
        Table Scan.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.

        FROM TABLE
            salesdetail
        EXISTS TABLE : nested iteration.
        Log Scan.
        Forward scan.
        Positioning at start of table.

        Run subquery 1 (at nesting level 1).
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.
        TO TABLE
            titles

NESTING LEVEL 1 SUBQUERIES FOR STATEMENT 4.

  QUERY PLAN FOR SUBQUERY 1 (at nesting level 1 and at line 23).

    Correlated Subquery.
    Subquery under an EXPRESSION predicate.


    STEP 1
        The type of query is SELECT.
        Evaluate Ungrouped SUM OR AVERAGE AGGREGATE.

        FROM TABLE
            salesdetail
        Nested iteration.
        Log Scan.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 2 Kbytes for data pages.
        With MRU Buffer Replacement Strategy for data pages.
```

I/O Size Messages

```
Using I/O size N Kbtyes for data pages.
Using I/O size N Kbtyes for index leaf pages.
```

The messages report the I/O sizes used in the query. The possible sizes are 2K, 4K, 8K, and 16K. If the table, index, LOB object, or database used in the query uses a data cache with large I/O pools, the optimizer can choose large I/O. It can choose to use one I/O size for reading index leaf pages, and a different size for data pages. The choice depends on the pool size available in the cache, the number of pages to be read, the cache bindings for the objects, and the cluster ratio for the table or index pages.

See Chapter 32, "Memory Use and Performance," for more information on large I/O and the data cache.

Cache Strategy Messages

```
With <LRU/MRU> Buffer Replacement Strategy for
data pages.
With <LRU/MRU> Buffer Replacement Strategy for
index leaf pages.
```

These messages indicate the cache strategy used for data pages and for index leaf pages. See "Overview of Cache Strategies" on page 3-21 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. If the query runs in parallel, the cost per thread is printed. System Administrators can use this value when setting compile-time resource limits. See "Total Actual I/O Cost Value" on page 16-4 for information on how cost is computed. If you are using **dbcc traceon(310)**, this value is the sum of the values in the FINAL PLAN output for the query.

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.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 16 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.

Total estimated I/O cost for statement 1 (at line 1): 1160.
```

For more information on creating resource limits, see Chapter 18, "Limiting Access to Server Resources," in the *System Administration Guide*.

showplan Messages for Parallel Queries

showplan reports information about parallel execution, showing which query steps are executed in parallel.

Table 17-4: showplan messages for parallel queries

Message	Explanation	See
`Executed in parallel by coordinating process and` *N* `worker processes.`	Indicates that a query is run in parallel, and shows the number of worker processes used.	Page 17-43
`Executed in parallel by` *N* `worker processes.`	Indicates the number of worker processes used for a query step.	Page 17-44
`Executed in parallel with a` *N*`-way hash scan.` `Executed in parallel with a` *N*`-way partition scan.`	Indicates the number of worker processes and the type of scan, hash-based or partition-based, for a query step.	Page 17-44

`Parallel work table merge.`
`Parallel network buffer merge.`
`Parallel result buffer merge.`
	Indicates the way in which the results of parallel scans are merged.	Page 17-44

Table 17-4: showplan messages for parallel queries (continued)

Message	Explanation	See
`Parallel data merge using` *`N`* `worker processes.`	Indicates that a merge join used a parallel data merge, and the number of worker processes used.	Page 17-47
`Serial data merge.`	Indicates that the merge join used a serial data merge.	Page 17-47
`AN ADJUSTED QUERY PLAN WILL BE USED FOR STATEMENT` *`N`* `BECAUSE NOT ENOUGH WORKER PROCESSES ARE AVAILABLE AT THIS TIME.` `ADJUSTED QUERY PLAN:`	Indicates that a runtime adjustment to the number of worker processes was required.	Page 17-47

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 12, "Parallel Query Optimization."

Coordinating Process Message

```
Executed in parallel by coordinating process and N
worker processes.
```

For each query that runs in parallel mode, **showplan** reports prints this message, indicating the number of worker processes used.

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.

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.”

Merge Messages

Results from the worker processes that process a query are merged using one of the following types of merge:

- Parallel worktable merge
- Parallel network buffer merge
- Parallel result buffer merge

Merge Message for Worktables

```
Parallel work table merge.
```

Grouped aggregate results from the worktables created by each worker process are merged 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).

    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.
        Forward scan.
        Positioning at start of table.
        Executed in parallel with a 2-way partition scan.
        Using I/O Size 16 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.
        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.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 16 Kbytes for data pages.
        With MRU Buffer Replacement Strategy for data pages.
```

See "Merge Join Messages" on page 17-25 for an example that uses parallel processing to perform sort-merge joins.

Merge Message for Buffer Merges

```
Parallel network buffer merge.
```

Unsorted, nonaggregate results returned by the worker processes are merged into a network buffer that is sent to the client. 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.
        Forward scan.
        Positioning at start of table.
        Executed in parallel with a 2-way partition scan.
        Using I/O Size 16 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.

        Parallel network buffer merge.
```

Merge Message for Result Buffers

```
Parallel result buffer merge.
```

Ungrouped aggregate results or unsorted, nonaggregate variable assignment results from worker processes are merged. Each worker process stores the aggregate in a result buffer. The result buffer merge produces a single value, ranging from zero-length (when the value is NULL) to the maximum length of a character string.

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.
        Forward scan.
```

```
    Positioning at start of table.
    Executed in parallel with a 2-way partition scan.
    Using I/O Size 16 Kbytes for data pages.
    With LRU Buffer Replacement Strategy for data pages.

    Parallel result buffer merge.

STEP 2
    The type of query is SELECT.
    Executed by coordinating process.
```

Data Merge Messages

```
Parallel data merge using N worker processes.
Serial data merge.
```

The data merge messages indicate whether a serial or parallel data merge was performed. If the merge is performed in parallel mode, the number of worker processes is also printed. For sample output, see “Merge Join Messages” on page 17-25.

Runtime 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:
```

showplan output displays this message and an adjusted query plan when fewer worker processes are available at runtime than the number specified by the optimized query plan.

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 17-5, include delimiters so that you can 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 17-5: showplan messages for subqueries

Message	Explanation	See
`Run subquery N (at nesting level N).`	This message indicates the point where the subquery runs. Subqueries are numbered in order for each side of a union.	Page 17-57
`NESTING LEVEL N SUBQUERIES FOR STATEMENT N.`	Shows the nesting level of the subquery.	Page 17-57
`QUERY PLAN FOR SUBQUERY N (at nesting level N and at line N).` `END OF QUERY PLAN FOR SUBQUERY N.`	These lines bracket **showplan** output for each subquery in a statement. Variables show the subquery number, the nesting level, and the input line.	Page 17-57
`Correlated Subquery.`	The subquery is correlated.	Page 17-57
`Non-correlated Subquery.`	The subquery is not correlated.	Page 17-57
`Subquery under an IN predicate.`	The subquery is introduced by **in**.	Page 17-58
`Subquery under an ANY predicate.`	The subquery is introduced by **any**.	Page 17-58
`Subquery under an ALL predicate.`	The subquery is introduced by **all**.	Page 17-58
`Subquery under an EXISTS predicate.`	The subquery is introduced by **exists**.	Page 17-58
`Subquery under an EXPRESSION predicate.`	The subquery is introduced by an expression, or the subquery is in the select list.	Page 17-58

`Evaluate Grouped <ONCE, ONCE-UNIQUE or ANY> AGGREGATE`
or
`Evaluate Ungrouped <ONCE, ONCE-UNIQUE or ANY> AGGREGATE`
	The subquery uses an internal aggregate.	Page 17-60
		
		Page 17-59

`EXISTS TABLE: nested iteration`	The query includes an **exists**, **in**, or **any** clause, and the subquery is flattened into a join.	Page 17-62

For information about how Adaptive Server optimizes certain types of subqueries by materializing results or by flattening the queries to joins, see “Subquery Optimization” on page 7-22. 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, or
- 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 subquery message delimiters. The following sections describe **showplan** output for flattened and materialized queries.

Flattened Queries

Adaptive Server can use one of several methods to flatten subqueries into joins. These methods are described in “Flattening in, any, and exists Subqueries” on page 7-23.

Subqueries Executed as Existence Joins

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 “Flattened Subqueries Executed as Existence Joins” on page 7-25.

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
        Forward scan.
        Positioning by key.
        Keys are:
            title  ASC
        Using I/O Size 16 Kbytes for index leaf pages.
        With LRU Buffer Replacement Strategy for index leaf pages.
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.

        FROM TABLE
            titleauthor
        EXISTS TABLE : nested iteration.
        Index : ta_ix
        Forward scan.
        Positioning by key.
        Index contains all needed columns. Base table will not be read.
        Keys are:
            title_id  ASC
        Using I/O Size 2 Kbytes for index leaf pages.
        With LRU Buffer Replacement Strategy for index leaf pages.
```

Subqueries Using Unique Reformatting

If there is not a unique index on *publishers.pub_id*, this query is flattened by selecting the rows from *publishers* into a worktable and then creating a unique clustered index. This process is called unique reformatting:

```
select title_id
from titles
where pub_id in
(select pub_id from publishers where state = "TX")
```

```
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
            publishers
        Nested iteration.
        Table Scan.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.
        TO TABLE
            Worktable1.

    STEP 2
        The type of query is SELECT.

        FROM TABLE
            Worktable1.
        Nested iteration.
        Using Clustered Index.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.

        FROM TABLE
            titles
        Nested iteration.
        Table Scan.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.
```

For more information, see “Flattened Subqueries Executed Using Unique Reformatting” on page 7-26.

Subqueries Using Duplicate Elimination

This query performs a regular join, selecting all of the rows into a worktable. In the second step, the worktable is sorted to remove duplicates. This process is called duplicate elimination:

```
select  title_id, au_id, au_ord
from titleauthor ta
where title_id in (select ta.title_id
    from titles t, salesdetail sd
    where t.title_id = sd.title_id
    and ta.title_id = t.title_id
    and type = 'travel' and qty > 10)
```

```
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
            salesdetail
            sd
        Nested iteration.
        Table Scan.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 16 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.

        FROM TABLE
            titles
            t
        Nested iteration.
        Using Clustered Index.
        Index : title_id_ix
        Forward scan.
        Positioning by key.
        Keys are:
            title_id  ASC
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.

        FROM TABLE
            titleauthor
            ta
        Nested iteration.
        Index : ta_ix
        Forward scan.
        Positioning by key.
        Keys are:
```

```
            title_id  ASC
        Using I/O Size 2 Kbytes for index leaf pages.
        With LRU Buffer Replacement Strategy for index leaf pages.
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.
        TO TABLE
            Worktable1.

    STEP 2
        The type of query is SELECT.
        This step involves sorting.

        FROM TABLE
            Worktable1.
        Using GETSORTED
        Table Scan.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 16 Kbytes for data pages.
        With MRU Buffer Replacement Strategy for data pages.
```

Materialized Queries

When Adaptive Server materializes subqueries, the query is executed in two steps:

1. The first step stores the results of the subquery in an internal variable or worktable.
2. The second step uses the internal variable or 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.
        Forward scan.
```

```
        Positioning at start of table.
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.
        TO TABLE
            Worktable1.

    STEP 2
        The type of query is SELECT.

        FROM TABLE
            titles
        Nested iteration.
        Table Scan.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 16 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.

        FROM TABLE
            Worktable1.
        EXISTS TABLE : nested iteration.
        Table Scan.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 16 Kbytes for data pages.
        With MRU Buffer Replacement Strategy for data pages.
```

The **showplan** message “EXISTS TABLE: nested iteration,” near the end of the output, shows that Adaptive Server performs an existence join.

Structure of Subquery *showplan* Output

When 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 nesting level *N* and 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 showing:

- The type of query (correlated or uncorrelated)
- The predicate type (IN, ANY, ALL, EXISTS, or EXPRESSION)

The output structure is shown in Figure 17-1, using the **showplan** output for this query:

```
select title_id
from titles
where total_sales > all (select total_sales
                         from titles
                         where type = 'business')
```

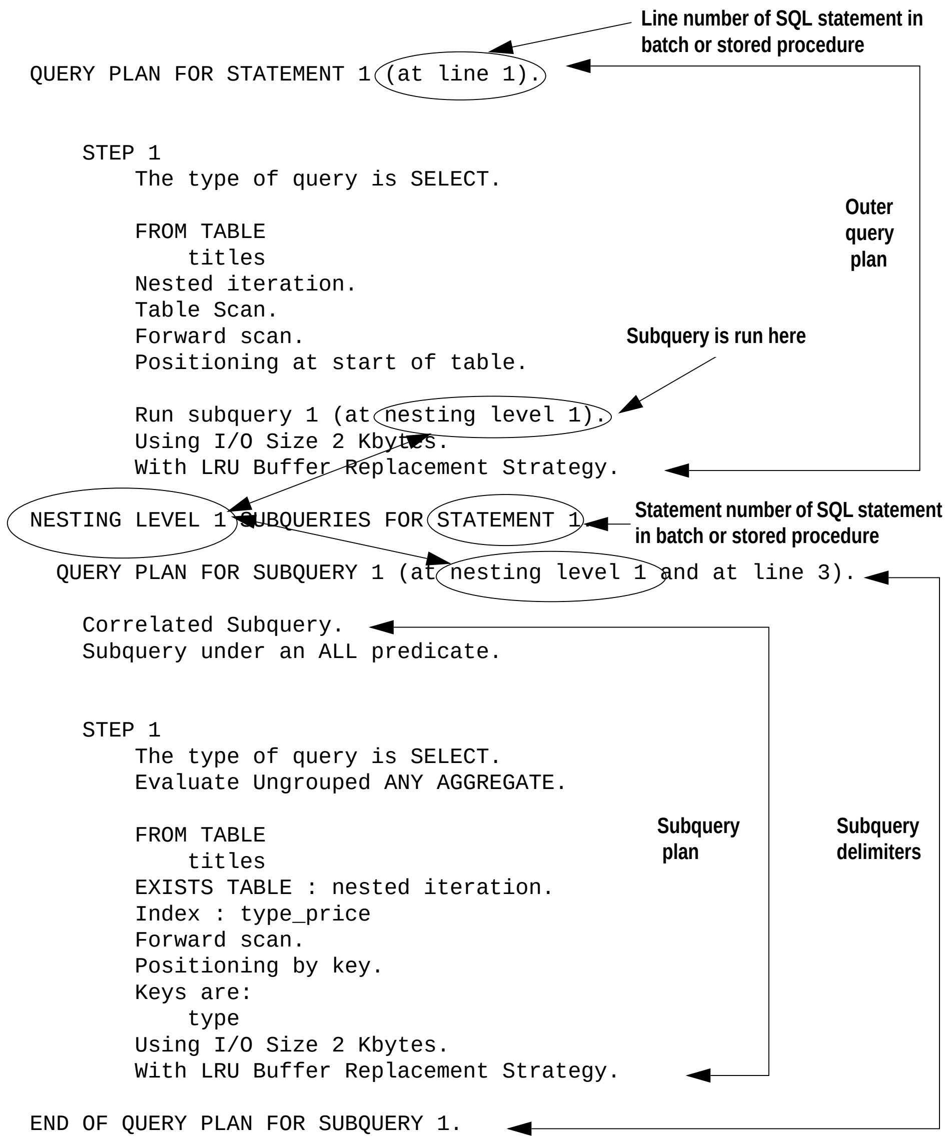


Figure 17-1: Subquery showplan output structure

Subquery Execution Message

```
Run subquery N (at nesting level N).
```

This message shows the place where the subquery execution takes place in the execution of the outer query. Adaptive Server executes the subquery at the point in the outer query where it need to be run least often.

The plan for this subquery appears later in the 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 nesting level N and
at line N).
```

This statement introduces the **showplan** output for a particular subquery at the nesting level indicated by the previous NESTING LEVEL message.

Line numbers to help you match **showplan** 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.
```

A subquery is either correlated or noncorrelated.

- A correlated subquery references a column in a table that is listed in the **from** list of the outer query. If the subquery is correlated, **showplan** includes the message "Correlated Subquery."
- A noncorrelated subquery can be evaluated independently of the outer query. Noncorrelated subqueries are sometimes materialized, so their **showplan** output does not include the normal subquery **showplan** messages.

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 17-6. Adaptive Server generates these aggregates internally—they are not part of Transact-SQL syntax and cannot be included in user queries.

Table 17-6: Internal subquery aggregates

Subquery Type	Aggregate	Effect
Quantified predicate	ANY	Returns TRUE or FALSE 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.

Messages for internal aggregates include “Grouped” when the subquery includes a **group by** clause and computes the aggregate for a group of rows, otherwise the messages include “Ungrouped”; the subquery 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 TRUE when a row from the subquery satisfies the conditions of the subquery predicate. It returns FALSE 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.
        Forward scan.
        Positioning at start of table.

        Run subquery 1 (at nesting level 1).
        Using I/O Size 16 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.

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.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 16 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.

  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
        Forward scan.
        Positioning by key.
        Keys are:
            title  ASC

        Run subquery 1 (at nesting level 1).
        Using I/O Size 16 Kbytes for index leaf pages.
```

```
        With LRU Buffer Replacement Strategy for index leaf pages.
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.

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.
        Table Scan.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.

  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.
        Forward scan.
```

```
        Positioning at start of table.

        Run subquery 1 (at nesting level 1).
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.

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 : pub_id_ix
        Forward scan.
        Positioning by key.
        Keys are:
            pub_id  ASC
        Using I/O Size 16 Kbytes for index leaf pages.
        With LRU Buffer Replacement Strategy for index leaf pages.
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.

  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 you can write queries that perform an existence test, for example, using **exists**, **in**, or =**any**. These queries are treated as if they were written with an **exists** clause. The following example shows 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.
        Forward scan.
        Positioning at start of table.

        Run subquery 1 (at nesting level 1).
        Using I/O Size 16 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.

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.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.

  END OF QUERY PLAN FOR SUBQUERY 1.
```

18 Tuning with *dbcc traceon*

This chapter describes the output of the **dbcc traceon(302, 310)** diagnostic tools. These tools can be used for debugging problems with query optimization. This chapter contains the following sections:

Tuning with *dbcc traceon(302)*

showplan tells you the final decisions that the optimizer makes about your queries. **dbcc traceon(302)** can often help you understand why the optimizer makes choices that seem incorrect. It can help you debug queries and decide whether to use certain options, like specifying an index or a join order for a particular query. It can also help you choose better indexes for your tables.

When you turn on **dbcc traceon(302)**, you eavesdrop on the optimizer as it examines query clauses and applies statistics for tables, search arguments, and join columns.

The output from this trace facility is more detailed than **showplan** and **statistics io** output, but it provides information about why the optimizer made certain query plan decisions. The query cost statistics printed by **dbcc traceon(302)** can help to explain, for example, why a table scan is chosen rather than an indexed access, why *index1* is chosen rather than *index2*, and so on.

dbcc traceon(310)

dbcc traceon(310) output can be extremely lengthy and is hard to understand without a thorough understanding of the optimizer. You often need to have your **showplan** output available as well to understand the join order, join type, and the join columns and indexes used. The most relevant parts of **dbcc traceon(310)** output, however, are the per-table total I/O estimates.

Invoking the *dbcc* Trace Facility

To start the **dbcc traceon(302)** trace facility, execute the following command from an **isql** batch, followed by the query or stored procedure that 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)
```

dbcc traceon(302) is often used in conjunction with **dbcc traceon(310)**, which provides more detail on the optimizer's join order decisions and final cost estimates. **dbcc traceon(310)** also prints a "Final plan" block at the end of query optimization. To enable this trace option also, use:

```
dbcc traceon(3604, 302, 310)
```

To turn off the output, use:

```
dbcc traceoff(3604, 302, 310)
```

See "dbcc traceon(310) and Final Query Plan Costs" on page 18-18 for information on **dbcc traceon(310)**.

General Tips for Tuning with *dbcc traceon(302)*

To get helpful output from **dbcc traceon(302)**, 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 tuning work. 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 cases, 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 first matches the search arguments to available indexes and statistics and estimates the number of rows and pages that qualify for each available index.

The most important item that you can verify using **dbcc traceon(302)** is that the optimizer is evaluating all possible **where** clauses included in the query. If a SARG clause is not included in the output, then the optimizer has determined it is not a valid search argument. 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.

Once all of the search arguments have been examined, each join combination is analyzed. If the optimizer is not choosing a join order that you expect, one of the first checks you should perform is to look for the sections of **dbcc traceon(302)** output that show join order costing: there should be two blocks of output for each join. If there is only one output for a given join, it means that the optimizer cannot consider using an index for the missing join order.

The most common reasons for “non-optimizable” clauses include:

- Use of functions, arithmetic, or concatenation on the column in a SARG, or on one of the join columns
- Datatype mismatches between SARGs and columns or between two columns in a join
- Numerics compared against constants that are larger than the definition of the column in a SARG, or joins between columns of different precision and scale

See “Search Arguments and Useful Indexes” on page 5-11 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 can to see when the optimizer uses actual statistics and when it uses default values for your search arguments.

Structure of *dbcc traceon(302)* Output

dbcc traceon(302) prints its output as the optimizer examines the clauses for each table involved in a query. The optimizer first examines all search clauses and determines the cost for each possible access method for the search clauses for each table in the query. It then examines each join clause and the cost of available indexes for the joins. **dbcc traceon(302)** output prints each search and join analysis as a block of output, delimited with a line of asterisks.

The search and join blocks each contain smaller blocks of information:

- A table information block, giving basic information on the table
- A block that shows the cost of a table scan
- A block that displays the clauses being analyzed
- A block for each index analyzed
- A block that shows the best index for the clauses in this section

For joins, each join order is represented by a separate block. For example, for these joins on *titles*, *titleauthor*, and *authors*:

```
where titles.title_id = titleauthor.title_id
    and authors.au_id = titleauthor.au_id
```

there is a block for each join, as follows:

- *titles*, *titleauthor*
- *titleauthor*, *titles*
- *titleauthor*, *authors*
- *authors*, *titleauthor*

Figure 18-1 shows the ordering and basic structure of the search and join blocks in **dbcc traceon(302)** output.

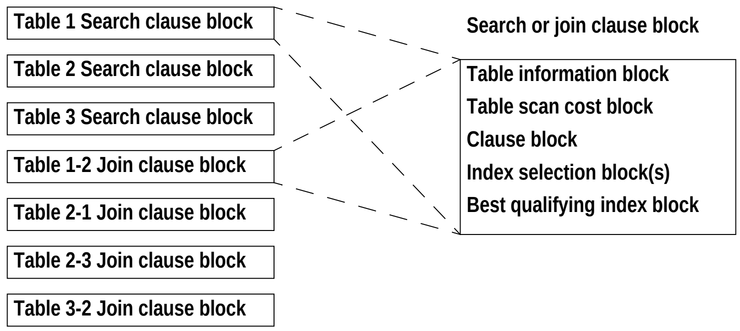


Figure 18-1: Structure of blocks of output in dbcc traceon(302)

Additional Blocks and Messages

Some queries generate additional blocks or messages in **dbcc traceon(302)** output, as follows:

- Queries that contain an **order by** clause contain additional blocks for displaying the analysis of indexes that can be used to avoid performing a sort. See "Sort Avert Messages" on page 18-9 for more information.
- Queries using transaction isolation level 0 (dirty reads) or updatable cursors on allpages-locked tables, where unique indexes are required, return a message like the following:

  ```
  Considering unique index 'au_id_ix', indid 2.
  ```

- Queries that specify an invalid prefetch size return a message like the following:

  ```
  Forced data prefetch size of 8K is not available.
  The largest available prefetch size will be used.
  ```

The Table Information Block

This sample output shows the table information block for a query on the *titles* table:

```
Beginning selection of qualifying indexes for table 'titles',
correlation name 't', varno = 0, objectid 208003772.
   The table (Datapages) has 5000 rows, 736 pages,
   Data Page Cluster Ratio 0.999990
   The table has 5 partitions.
   The largest partition has 211 pages.
   The partition skew is 1.406667.
```

Identifying the Table

The first two lines identify the table, giving the table name, the correlation name (if one was used in the query), a *varno* value that identifies the order of the table in the **from** clause, and the object ID for the table.

In the query, *titles* is specified using “t” as a correlation name, as in:

```
from titles t
```

The correlation name is included in the output only if a correlation name was used in the query. The correlation name is especially useful when you are trying to analyze the output from subqueries or queries doing self-joins on a table, such as:

```
from sysobjects o1, sysobjects o2
```

Basic Table Data

The next lines of output provide basic data about the table: the locking scheme, the number of rows, and the number of pages in the table. The locking scheme is one of: Allpages, Datapages, or Datarows.

Cluster Ratio

The next line prints the data page cluster ratio for the table.

Partition Information

The following lines are included only for partitioned tables. They give the number of partitions, plus the number of pages in the largest partition, and the skew:

```
The table has 5 partitions.
The largest partition has 211 pages.
The partition skew is 1.406667.
```

This information is useful if you are tuning parallel queries, because:

- Costing for parallel queries is based on the cost of accessing the table's largest partition.
- The optimizer does not choose a parallel plan if the partition skew is 2.0 or greater.

See Chapter 11, "Introduction to Parallel Query Processing," for more information on parallel query optimization.

The Base Cost Block

The optimizer determines the cost of a table scan as a first step. It also displays the caches used by the table, the availability of large I/O, and the cache replacement strategy.

The following output shows the base cost for the *titles* table:

```
Table scan cost is 5000 rows, 748 pages,
   using data prefetch (size 16K I/O),
   in data cache 'default data cache' (cacheid 0) with LRU replacement
```

If the cache used by the query has only a 2K pool, the prefetch message is replace by:

```
using no data prefetch (size 2K I/O)
```

Concurrency Optimization Message

For very small data-only-locked tables, the following message may be included in this block:

```
If this table has useful indexes, a table scan
will not be considered because concurrency
optimization is turned ON for this table.
```

For more information, see "Concurrency Optimization for Small Tables" on page 20-16.

The Clause Block

The clause block prints the search clauses and join clauses that the optimizer considers while it estimates the cost of each index on the table. Search clauses for all tables are analyzed first, and then join clauses.

Search Clause Identification

For search clauses, the clause block prints each of the search clauses that the optimizer can use. The list should be compared carefully to the clauses that are included in your query. If query clauses are not listed, it means that the optimizer did not evaluate them because it cannot use them. For example, this set of clauses on the *titles* table:

```
where type = "business"
    and title like "B%"
    and total_sales > 12 * 1000
```

produces this list of optimizable search clauses, with the table names preceding the column names:

```
Selecting best index for the SEARCH CLAUSE:
        titles.title < 'C'
        titles.title >= 'B'
        titles.type = 'business'
        titles.total_sales > 12000
```

Notice that the **like** has been expanded into a range query, searching for >= ‘B’ and <‘C’. All of the clauses in the SQL statement are included in the **dbcc traceon(302)** output, and can be used to help optimize the query. If search argument transitive closure and predicate factoring have added optimizable search arguments, these are included in this costing block also. See “Search Arguments and Useful Indexes” on page 5-11 for more information.

When Search Clauses Are Not Optimizable

The following set of clauses on the *authors* table includes the **substring** function on the *au_fname* column:

```
where substring(au_fname,1,4) = "Fred"
    and city = "Miami"
```

Due to the use of the **substring** function on a column name, the set of optimizable clauses does not include the **where** clause on the *au_fname* column:

```
Selecting best index for the SEARCH CLAUSE:
    authors.city = 'Miami'
```

Values Unknown At Optimize Time

For values that are not known at optimize time, **dbcc traceon(302)** prints "unknown-value." For example, this clause uses the **getdate** function:

```
where pubdate > getdate()
```

It produces this message in the search clause list:

```
titles.pubdate > unknown-value
```

Join Clause Identification

Once all of the search clauses for each table have been analyzed, the join clauses are analyzed and optimized. Each table is analyzed in the order listed in the **from** clause. **dbcc traceon(302)** prints the operator and table and column names, as shown in this sample output of a join between *titleauthor* and *titles*, during the costing of the *titleauthor* table:

```
Selecting best index for the JOIN CLAUSE:
        titleauthor.title_id = titles.title_id
```

The table currently undergoing analysis is always printed on the left in the join clause output. When the *titles* table is being analyzed, *titles* is printed first:

```
Selecting best index for the JOIN CLAUSE:
        titles.title_id = titleauthor.title_id
```

If you expect an index for a join column to be used, and it is not, check for the JOIN CLAUSE output with the table as the leading table. If it is not included in the output, check for datatype mismatches on the join columns.

Sort Avert Messages

If the query includes an **order by** clause, additional messages are displayed. The optimizer checks to see if an index matches the ordering required by the **order by** clause, to avoid incurring sort costs for the query.

This message is printed for search clauses:

```
Selecting best index for the SEARCH SORTAVERT CLAUSE:
    titles.type = 'business'
```

The message for joins shows the column under consideration first. This message is printed while the optimizer analyzes the *titles* table:

```
Selecting best index for the JOIN SORTAVERT CLAUSE:
    titles.title_id = titleauthor.title_id
```

At the end of the block for the search or join clause, one of two messages is printed, depending on whether an index exists that can be used to avoid performing a sort step. If no index is available, this message is printed:

```
No sort avert index has been found for table
'titles' (objectid 208003772, varno = 0).
```

If an index can be used to avoid the sort step, the sort-avert message includes the index ID, the number of pages that need to be accessed, and the number of rows to be returned for each scan. This is a typical message:

```
The best sort-avert index is index 3, costing 9
pages and generating 8 rows per scan.
```

This message does not mean that the optimizer has decided to use this index. It means simply that, if this index is used, it does not require a sort. If you expect an index to be used to avoid a sort, and you see the "No sort avert index" message, check the **order by** clauses in the query for the use of **asc** and **desc** to request ascending and descending ordering, and check the ordering specifications for the index. For more information, see "Query Costing for Queries Using order by" on page 6-16.

The Column Block

This section prints the selectivity of each optimizable search argument or join clause. Selectivity is used to estimate the number of matching values for a search clause or join clause.

The optimizer uses column statistics, if they exist and if the value of the search argument is known at optimize time. If not, the optimizer uses default values.

Selectivities When Statistics Exist and Values Are Known

This shows the selectivities for a search clause on the *title* column, when an index exists for the column:

```
Estimated selectivity for title,
   selectivity = 0.001077, upper limit = 0.060200.
```

For equality search arguments where the value falls in a range cell:

- The selectivity is the “Range cell density” displayed by **optdiag**.
- The upper limit is the weight of the histogram cell.

If the value matches a frequency cell, the selectivity and upper limit are the weight of that cell.

For range queries, the upper limit is the sum of the weights of all histogram cells that contain values in the range. The upper limit is used only in cases where interpolation yields a selectivity that is greater than the upper limit.

The upper limit is not printed when the selectivity for a search argument is 1.

For join clauses, the selectivity is the “Total density” displayed by **optdiag**.

When the Optimizer Uses Default Values

The optimizer uses default values for selectivity:

- When the value of a search argument is not known at the time the query is optimized
- For search arguments where no statistics are available

In both of these cases, the optimizer uses different default values, depending on the operators used in the query clause.

Unknown Values

Unknown values include variables that are set in the same batch as the query and values set within a stored procedure. This message indicates an unknown value for a column where statistics are available and the equality (=) operator is used:

```
SARG is a local variable or the result of a function or an
expression, using the total density to estimate selectivity.
```

Similar messages are printed for open-ended range queries and queries using **between**.

If No Statistics Are Available

If no statistics are available for a column, a message indicates the default selectivity that will be used. This message is printed for an open-ended range query on the *total_sales* table:

```
No statistics available for total_sales,
using the default range selectivity to estimate selectivity.

Estimated selectivity for total_sales,
      selectivity = 0.330000.
```

See "Default Values for Search Arguments" on page 5-17 for the default values used for search arguments and "When Statistics Are Not Available for Joins" on page 5-20 for the default values used for joins.

You may be able to improve optimization for queries where default values are used frequently, by creating statistics on the columns. See "Creating and Updating Column Statistics" on page 10-4.

Out-of-Range Messages

Out-of-range messages are printed when a search argument is out of range of the values included in the histogram for an indexed column.

The following clause searches for a value greater than the last *title_id*:

```
where title_id > "Z"
```

dbcc traceon(302) prints:

```
Estimated selectivity for title_id,
   selectivity = 0.000000, upper limit = 0.000000.
Lower bound search value ''Z'' is greater than the largest value
in sysstatistics for this column.
```

For a clause that searches for a value that is less than the first key value in an index, **dbcc traceon(302)** prints:

```
Estimated selectivity for title_id,
    selectivity = 0.000000, upper limit = 0.000000.
Upper bound search value ''B'' is less than the smallest value
in sysstatistics for this column.
```

If the equality operator is used instead of a range operator, the messages read:

```
Estimated selectivity for title_id,
   selectivity = 0.000000, upper limit = 0.000000.
Equi-SARG search value ''Zebracode'' is greater than the largest
value in sysstatistics for this column.
```

or:

```
Estimated selectivity for title_id,
   selectivity = 0.000000, upper limit = 0.000000.
Equi-SARG search value ''Applepie'' is less than the smallest
value in sysstatistics for this column.
```

These messages may simply indicate that the search argument used in the query is out of range for the values in the table. In that case, no rows are returned by the query. However, if there are matching values for the out-of-range keys, it may indicate that it is time to run **update statistics** on the table or column, since rows containing these values must have been added since the last time the histogram was generated.

There is a special case for search clauses using the **>=** operator and a value that is less than or equal to the lowest column value in the histogram. For example, if the lowest value in an integer column is 20, this clause:

```
where col1 >= 16
```

produces this message:

```
Lower bound search condition '>= 16' includes all values in this
column.
```

For these cases, the optimizer assumes that all non-null values in the table qualify for this search condition.

“Disjoint Qualifications” Message

The “disjoint qualifications” message often indicates a user error in specifying the search clauses. For example, this query searches for a range where there could be no values that match both of the clauses:

```
where advance > 10000
and advance < 1000
```

The selectivity for such a set of clauses is always 0.0, meaning that no rows match these qualifications, as shown in this output:

```
Estimated selectivity for advance,
   disjoint qualifications, selectivity is 0.0.
```

Forcing Messages

dbcc traceon(302) prints messages if any of the index, I/O size, buffer strategy, or parallel force options are included for a table or if an abstract plan specifying these scan properties was used to optimize the query. Here are sample messages for a query using an abstract plan:

```
For table ‘titles’:
User forces index 2 (index name = type_price_ix)
User forces index and data prefetch of 16K
User forces MRU buffer replacement strategy on
index and data pages
User forces parallel strategy.  Parallel Degree = 3
```

Unique Index Messages

When a unique index is being considered for a join or a search argument, the optimizer knows that the query will return one row per scan. The message includes the index type, the string “returns 1 row,” and a page estimate, which includes the number of index levels, plus one data page:

```
Unique clustered index found, returns 1 row, 2
pages
```

```
Unique nonclustered index found, returns 1 row, 3
pages
```

Other Messages in the Column Block

If the statistics for the column have been modified using **optdiag**, **dbcc traceon(302)** prints:

```
Statistics for this column have been edited.
```

If the statistics result from an upgrade of an earlier version of the server or from loading a database from an pre-11.9 version of the server, **dbcc traceon(302)** prints:

```
Statistics for this column were obtained from
upgrade.
```

If this message appears, run **update statistics** for the table or index.

The Index Selection Block

While costing index access, **dbcc traceon(302)** prints a set of statistics for each useful index. This index block shows statistics for an index on *au_lname* in the *authors* table:

```
Estimating selectivity of index 'au_names_ix', indid 2
   scan selectivity 0.000936, filter selectivity 0.000936
   5 rows, 3 pages, index height 2,
   Data Row Cluster Ratio 0.990535,
   Index Page Cluster Ratio 0.538462,
   Data Page Cluster Ratio 0.933579
```

Scan and Filter Selectivity Values

The index selection block includes, a scan selectivity value and a filter selectivity value. In the example above, these values are the same (0.000936). For queries that specify search arguments on multiple columns, these values are different when the search arguments include the leading key, and some other index key that is not part of a prefix subset. That is, if the index is on columns A, B, C, D, a query specifying search arguments on A, B, and D will have different scan and filter selectivities. The two selectivities are used for estimating costs at different levels:

	Scan Selectivity	Filter Selectivity
Used to estimate:	Number of index rows and leaf-level pages to be read	Number of data pages to be accessed
Determined by:	Search arguments on leading columns in the index	All search arguments on the index under consideration. even if they are not part of the prefix subset for the index

How Scan and Filter Selectivity Can Differ

This statement creates a composite index on *titles*:

```
create index composite_ix
on titles (pub_id, type, price)
```

Both of the following clauses can be used to position the start of the search and to limit the end point, since the leading columns are specified:

```
where pub_id = "P099"
where pub_id = "P099" and type = "news"
```

The first example requires reading all the index pages where *pub_id* equals “P099”, while the second reads only the index pages where both conditions are true. In both cases, these queries need to read the data rows for each of the index rows that are examined, so the scan and filter selectivity are the same.

In the following example, the query needs to read all of the index leaf-level pages where *pub_id* equals “P099”, as in the queries above. But in this case, Adaptive Server examines the value for *price*, and needs to read only those data pages where the *price* is less than $50:

```
where pub_id = "P099" and price < $50
```

In this case, the scan and filter selectivity differ. If column-level statistics exist for *price*, the optimizer combines the column statistics on *pub_id* and *price* to determine the filter selectivity, otherwise the filter selectivity is estimated using the default range selectivity.

In the **dbcc traceon(302)** output below, the selectivity for the *price* column uses the default value, 0.33, for an open range. When combined with the selectivity of 0.031400 for *pub_id*, it yields the filter selectivity of 0.010362 for *composite_ix*:

```
Selecting best index for the SEARCH CLAUSE:
      titles.price < 50.00
      titles.pub_id = 'P099'

Estimated selectivity for pub_id,
   selectivity = 0.031400, upper limit = 0.031400.

No statistics available for price,
using the default range selectivity to estimate selectivity.

Estimated selectivity for price,
   selectivity = 0.330000.

Estimating selectivity of index 'composite_ix', indid 6
   scan selectivity 0.031400, filter selectivity 0.010362
   52 rows, 57 pages, index height 2,
   Data Row Cluster Ratio 0.013245,
   Index Page Cluster Ratio 1.000000,
   Data Page Cluster Ratio 0.100123
```

Other Information in the Index Selection Block

The index selection block prints out an estimate of the number of rows that would be returned if this index were used and the number of pages that would need to be read. The index height is also included. For a single-table query, this information is basically all that is needed for the optimizer to choose between a table scan and the available indexes. For joins, this information is used later in optimization to help determine the cost of various join orders.

The three cluster ratio values for the index are printed, since estimates for the number of pages depend on cluster ratios.

If the index covers the query, this block includes the line:

```
Index covers query.
```

This message indicates that the data pages of the table do not have to be accessed if this index is chosen.

The Best Access Block

The final section for each SARG or join block for a table shows the best qualifying index for the clauses examined in the block.

When search arguments are being analyzed, the best access block looks like:

```
The best qualifying  index is 'pub_id_ix' (indid 5)
   costing 153 pages,
   with an estimate of 168 rows to be returned per scan of the table,
   using index prefetch (size 16K I/O) on leaf pages,
   in index cache 'default data cache' (cacheid 0) with LRU replacement
   using no data prefetch (size 2K I/O),
   in data cache 'default data cache' (cacheid 0) with LRU replacement
Search argument selectivity is 0.033539.
```

If no useful index is found, the final block looks like:

```
The best qualifying access is a table scan,
   costing 621 pages,
   with an estimate of 1650 rows to be returned per scan of the table,
   using data prefetch (size 16K I/O),
   in data cache 'default data cache' (cacheid 0) with LRU replacement
Search argument selectivity is 0.330000.
```

For joins, there are two best access blocks when a merge join is considered during the join-costing phase, one for nested-loop join cost, and one for merge-join cost:

```
The best qualifying Nested Loop join index is 'au_city_ix' (indid 4)
   costing 6 pages,
   with an estimate of 4 rows to be returned per scan of the table,
   using index prefetch (size 16K I/O) on leaf pages,
   in index cache 'default data cache' (cacheid 0) with LRU replacement
   using no data prefetch (size 2K I/O),
   in data cache 'default data cache' (cacheid 0) with LRU replacement
Join selectivity is 0.000728.


The best qualifying Merge join index is 'au_city_ix' (indid 4)
   costing 6 pages,
   with an estimate of 4 rows to be returned per scan of the table,
   using no index prefetch (size 2K I/O) on leaf pages,
   in index cache 'default data cache' (cacheid 0) with LRU replacement
   using no data prefetch (size 2K I/O),
   in data cache 'default data cache' (cacheid 0) with LRU replacement
Join selectivity is 0.000728.
```

Note that the output in this block estimates the number of “rows to be returned per scan of the table.” At this point in query optimization, the join order has not yet been chosen. If this table is the outer table, the total cost of accessing the table is 6 pages, and it is estimated to return 4 rows. If this query is an inner table of a nested-loop join, with a cost of 6 pages each time, each access is estimated to return 4 rows. The number of times the table will be scanned depends on the number of estimated qualifying rows for the other table in the join.

If no index qualifies as a possible merge-join index, **dbcc traceon(302)** prints:

```
If this access path is selected for merge join, it will be sorted
```

dbcc traceon(310) and Final Query Plan Costs

The end of each search clause and join clause block prints the best index for the search or join clauses in that particular block. If you are concerned only about the optimization of the search arguments, **dbcc traceon(302)** output has probably provided the information you need.

The choice of the best query plan also depends on the join order for the tables, which is the next step in query optimization after the index costing step completes. **dbcc traceon(310)** provides information about the join order selection step.

It starts by showing the number of tables considered at a time during a join. This message shows three-at-a-time optimization, with the default for **set table count**, and a 32-table join:

```
QUERY IS CONNECTED
Number of tables in join: 32
Number of tables considered at a time: 3
Table count setting: 0 (default value used)
```

dbcc traceon(310) prints the first plan that the optimizer considers, and then each cheaper plan, with the heading "NEW PLAN." To see all of the plans, use **dbcc traceon(317).** It prints each plan considered, with the heading "WORK PLAN." This may produce an extremely large amount of output, especially for queries with many tables, many indexes, and numerous query clauses. If you use dbcc **traceon(317)**, also use **dbcc traceon(3604)** and direct the output to a file, rather than to the server's error log to avoid filling up the error log device.

dbcc traceon(310) or **(317)** prints the join orders being considered as the optimizer analyzes each of the permutations. It uses the *varno*, representing the order of the tables in the **from** clause. For example, for the first permutation, it prints:

```
 0 - 1 - 2 -
```

This is followed by the cost of joining the tables in this order. The permutation order for subsequent join orders follows, with "NEW PLAN" and the analysis of each table for the plan appearing whenever a cheaper plan is found. Once all plans have been examined, the final plan is repeated, with the heading "FINAL PLAN". This is the plan that Adaptive Server uses for the query.

Flattened Subquery Join Order Message

For some flattened subqueries, certain join orders are possible only if a sort is later used to remove duplicate results. When one of these join orders is considered, the following message is printed right after the join permutation order is printed:

```
2 - 0 - 1 -

This join order created while converting an exists
join to a regular join, which can happen for
subqueries, referential integrity, and select
distinct.
```

For more information on subqueries and join orders, see "Flattened Subqueries Using Duplicate Elimination d" on page 7-27.

Worker Process Information

Just before printing final plan information, **dbcc traceon(310)** prints the parallel configuration parameters and session level settings in effect when the command was run.

```
PARALLEL:
      number of worker processes = 20
      max parallel degree = 10
      min(configured,set) parallel degree = 10
      min(configured,set) hash scan parallel degree = 3
```

If session-level limits or simulated statistics in effect when the query is optimized, those values are shown in the output.

Final Plan Information

The plan chosen by the optimizer is displayed in the final plan block. Information about the cost of each table is printed; the output starts from the outermost table in the join order.

```
select pub_name, au_lname, price
from titles t, authors a, titleauthor ta,
             publishers p
where t.title_id = ta.title_id
    and a.au_id = ta.au_id
    and p.pub_id = t.pub_id
    and type = 'business'
    and price < $25
```

```
FINAL PLAN (total cost = 3909)

varno=0 (titles) indexid=1 (title_id_ix)
path=0xd6b25148 pathtype=pll-mrgscan-outer
method=NESTED ITERATION
scanthreads=3
outerrows=1 outer_wktable_pgs=0 rows=164 joinsel=1.000000
jnpgs_per_scan=3 scanpgs=623
data_prefetch=YES data_iosize=16 data_bufreplace=LRU
scanlio_perthrd=211 tot_scanlio=633 scanpio_perthrd=116 tot_scanpio=346
outer_srtmrglio=0 inner_srtmrglio=0
corder=1

varno=2 (titleauthor) indexid=3 (ta_ix)
path=0xd6b20000 pathtype=pll-mrgscan-inner
method=FULL MERGE JOIN
scanthreads=3 mergethreads=3
outerrows=164 outer_wktable_pgs=0 rows=243 joinsel=0.000237
jnpgs_per_scan=2 scanpgs=87
index_prefetch=YES index_iosize=16 index_bufreplace=LRU
```

```
scanlio_perthrd=29 total_scanlio=87 scanpio_perthrd=29 tot_scanpio=87
outer_srtmrglio_perthrd=0 tot_outer_srtmrglio=0
inner_srtmrglio_perthrd=0 tot_inner_srtmrglio=0
corder=2

varno=1 (authors) indexid=3 (au_id_ix)
path=0xd6b20318 pathtype=join
method=NESTED ITERATION
scanthreads=1
outerrows=243 rows=243 joinsel=0.000200 jnpgs_per_scan=3
index_prefetch=NO index_iosize=2 index_bufreplace=LRU
data_prefetch=NO data_iosize=2 data_bufreplace=LRU
scanlio=82 scanpio=9
corder=1

jnvar=2 refcost=0 refpages=0 reftotpages=0 ordercol[0]=1  ordercol[1]=1

varno=3 (publishers) indexid=0 ()
path=0xd6b1f150 pathtype=sclause
method=SORT MERGE JOIN
scanthreads=1
outerrows=243 outer_wktable_pgs=7 rows=243 joinsel=0.033333
jnpgs_per_scan=1 scanpgs=3
data_prefetch=NO data_iosize=2 data_bufreplace=LRU
scanlio=3 scanpio=3
outer_srtmrglio_perthrd=88 tot_outer_srtmrglio=250
inner_srtmrglio_perthrd=31 tot_inner_srtmrglio=30
corder=0

Sort-Merge Cost of Inner = 98
Sort-Merge Cost of Outer = 344
```

For the **showplan** output for the same query, see “Merge Join Messages” on page 17-25.

Table 18-1 shows the meaning of the values in the output.

Table 18-1: dbcc traceon(310) output

Label	Information Provided
varno	Indicates the table order in the **from** clause, starting with 0 for the first table. The table name is provided in parentheses.
indexid	The index ID, followed by the index name, or 0 for a table scan.
pathtype	The access method for this table. See Table 18-2.

Table 18-1: dbcc traceon(310) output (continued)

Label	Information Provided
method	The method used for the scan or join: • NESTED ITERATION • NESTED ITERATION with Tuple Filtering • REFORMATTING • REFORMATTING with Unique Reformatting • OR OPTIMIZATION • SORT MERGE JOIN • RIGHT MERGE JOIN • LEFT MERGE JOIN • FULL MERGE JOIN
scanthreads	Number of worker processes to be used for the scan of this table.
mergethreads	Number of threads to use for a parallel data merge, for a sort-merge join.
outerrows	Number of rows that qualify from the outer tables in the query or 1, for the first table in the join order.
outer_wktable_pgs	For a merge join, the number of pages in the worktable that is outer to this table, or tables in a full-merge join.
rows	Number of rows estimated to qualify in this table or as a result of this join. For a parallel query, this is the maximum number of rows per worker process.
joinsel	The join selectivity.
jnpgs_per_scan	Number of index and data pages to be read for each scan.
scanpgs	The total number of index and data pages to be read for the table.
index_prefetch	YES if large I/O will be used on index leaf pages (not printed for table scans and allpages-locked table clustered index scans).
index_iosize	The I/O size to be used on the index leaf pages (not printed for table scans and allpages-locked table clustered index scans).
index_bufreplace	The buffer replacement strategy to be used on the index leaf pages (not printed for table scans and allpages-locked table clustered index scans).
data_prefetch	YES if large I/O will be used on the data pages; NO if large I/O will not be used (not printed for covered scans).
data_iosize	The I/O size to be used on the data pages (not printed for covered scans).

Table 18-1: dbcc traceon(310) output (continued)

Label	Information Provided
data_bufreplace	The buffer replacement strategy to be used on the data pages (not printed for covered scans).
scanlio	Estimated total logical I/O for a serial query.
scanpio	Estimated total physical I/O for a serial query.
scanlio_perthrd	Estimated logical I/O per thread, for a parallel query.
tot_scanlio	Estimated total logical I/O, for a parallel query.
scanpio_perthrd	Estimated physical I/O per thread, for a parallel query.
tot_scanpio	Estimated total physical I/O, for a parallel query.
outer_srtmrglio_perthrd	Estimated logical I/O on the outer table to perform the sort-merge, per thread.
tot_outer_srtmrglio	Estimated total logical I/O on the outer table to perform a sort-merge.
inner_srtmrglio_perthrd	Estimated logical I/O on the inner table to perform a sort-merge join, per thread.
tot_inner_srtmrglio	Estimated total logical I/O on the inner table to perform a sort-merge join.
corder	The order of the column used as a search argument or join key.
jnvar	The *varno* of the table to which this table is being joined, for second and subsequent tables in a join.
refcost	The total cost of reformatting, when reformatting is considered as an access method.
refpages	The number of pages read in each scan of the table created for formatting. Included for the second and subsequent tables in the join order.
reftotpages	The number of pages in the table created for formatting. Included for the second and subsequent tables in the join order.
ordercol[0]	The order of the join column from the inner table.
ordercol[1]	The order of the join column from the outer table.

Table 18-2 shows the access methods that correspond to the *pathtype* information in the **dbcc traceon(310)** output.

Table 18-2: pathtypes in dbcc traceon(310) output

pathtype	**Access method**
sclause	Search clause
join	Join
orstruct	**or** clause
join-sort	Join, using a sort-avert index
sclause-sort	Search clause, using a sort-avert index
pll-sarg-nc	Parallel index hash scan on a search clause
pll-join-nc	Parallel index hash scan on a join clause
pll-sarg-cl	Parallel clustered index scan on a search clause
pll-join-cl	Parallel clustered index scan on a join
pll-sarg-cp	Parallel partitioned clustered index scan on a search clause
pll-join-cp	Parallel partitioned clustered index scan on a join clause
pll-partition	Parallel partitioned table scan
pll-nonpart	Parallel nonpartitioned table scan
pll-mrg-scan-inner	Parallel sort-merge join, with this table as the inner table
pll-mrg-scan-outer	Parallel sort-merge join, with this table as the outer table

Sort-Merge Costs

If the query plan includes a sort-merge join, the cost of creating the worktables and sorting them are printed. These messages include the total cost that is added to the query cost:

```
Sort-Merge Cost of Inner = 538
Sort-Merge Cost of Outer = 5324
```

These are the total costs of performing the sort-merge work, representing the logical I/O on the worktables multiplied by 2.

19 Statistics Tables and Displaying Statistics with *optdiag*

This chapter explains how statistics are stored and displayed. Topics include:

For more information on managing statistics, see Chapter 10, "Managing Statistics to Improve Performance."

System Tables That Store Statistics

The *systabstats* and *sysstatistics* tables store statistics for all tables, indexes, and any unindexed columns for which you have explicitly created statistics. In general terms:

- *systabstats* stores information about the table or index as an object, that is, the size, number of rows, and so forth. It is updated by query processing, data definition language, and **update statistics** commands.
- *sysstatistics* stores information about the values in a specific column. It is updated by data definition language and **update statistics** commands.

For more information, see "Effects of SQL Commands on Statistics" on page 19-32.

systabstats Table

The *systabstats* table contains basic statistics for tables and indexes, for example:

- Number of data pages for a table, or the number of leaf level pages for an index
- Number of rows in the table
- Height of the index

- Average length of data rows and leaf rows
- Number of forwarded and deleted rows
- Number of empty pages
- Statistics to increase the accuracy of I/O cost estimates, including cluster ratios, the number of pages that share an extent with an allocation page, and the number of OAM and allocation pages used for the object
- Stopping points for the **reorg** command so that it can resume processing

systabstats stores one row for each table and nonclustered index in the database. The storage for clustered index information depends on the locking scheme for the table:

- If the table is a data-only-locked table, *systabstats* stores an additional row for a clustered index.
- If the table is an allpages-locked table, the data pages are treated as the leaf level of the index, so the *systabstats* entry for a clustered index is stored in the same row as the table data. The *indid* column for clustered indexes on allpages-locked tables is always 1.

See *systabstats* in the *Adaptive Server Reference Manual* for more information.

sysstatistics Table

The *sysstatistics* table stores one or more rows for each indexed column on a user table. In addition, it can store statistics for unindexed columns.

The first row for each column stores basic statistics about the column, such as the density for joins and search arguments, the selectivity for some operators, and the number of steps stored in the histogram for the column. If the index has multiple columns, or if you specify multiple columns when you generate statistics for unindexed columns, there is a row for each prefix subset of columns. For more information on prefix subsets, see "Column Statistics" on page 19-11.

Additional rows store histogram data for the leading column. Histograms do not exist if indexes were created before any data was inserted into a table (run **update statistics** after inserting data to generate the histogram.) See "Histogram Displays" on page 19-15 for more information.

See *sysstatistics* in the *Adaptive Server Reference Manual* for more information.

Viewing Statistics with the *optdiag* Utility

The **optdiag** utility displays statistics from the *systabstats* and *sysstatistics* tables. **optdiag** can also be used to update *sysstatistics* information. Only a System Administrator can run **optdiag**.

optdiag Syntax

The syntax for **optdiag** is:

```
optdiag  [ binary ] [simulate ] statistics
       { -i input_file |
       database[.owner[.[table[.column]]]]
           [-o output_file] }
       [-U username] [-P password]
       [-I interfaces_file]
       [-S server]
       [-v] [-h] [-s] [-Tflag_value]
       [-z language] [-J client_charset]
       [-a display_charset]
```

You can use **optdiag** to display statistics for an entire database, for a single table and its indexes and columns, or for a particular column. To display statistics for all user tables in the *pubtune* database, placing the output in the *pubtune.opt* file, use the following command:

```
optdiag statistics pubtune -Usa -Ppasswd
-o pubtune.opt
```

This command displays statistics for the *titles* table and for any indexes on the table:

```
optdiag statistics pubtune..titles -Usa -Ppasswd
        -o titles.opt
```

See the *Utility Programs* manual for your platform for more information on the **optdiag** command. The follow sections provide information about the output from **optdiag**.

optdiag Header Information

After printing the version information for **optdiag** and Adaptive Server, **optdiag** prints the server name and summarizes the arguments used to display the statistics.

The header of the **optdiag** report lists the objects described in the report:

```
Server name:                           "test_server"

Specified database:                    "pubtune"
Specified table owner:                 not specified
Specified table:                       "titles"
Specified column:                      not specified
```

Table 19-1 describes the output.

Table 19-1: Table and column information

Row Label	Information Provided
Server name	The name of the server, as stored in the *@@servername* variable. You must use **sp_addserver**, and restart the server for the server name to be available in the variable.
Specified database	Database name given on the **optdiag** command line.
Specified table owner	Table owner given on the **optdiag** command line.
Specified table	Table name given on the **optdiag** command line.
Specified column	Column name given on the **optdiag** command line.

Table Statistics

This **optdiag** section reports basic statistics for the table.

Sample Output for Table Statistics

```
Table owner:                        "dbo"

Statistics for table:               "titles"

     Data page count:               662
     Empty data page count:         10
     Data row count:                4986.0000000000000000
     Forwarded row count:           18.0000000000000000
     Deleted row count:             87.0000000000000000
     Data page CR count:            86.0000000000000000
     OAM + allocation page count:   5
     First extent data pages:       3
     Data row size:                 238.8634175691937287

  Derived statistics:
     Data page cluster ratio:       0.9896907216494846
```

Table 19-2 describes the rows in the report.

Table 19-2: Table statistics

Row Label	Information Provided
Table owner	Name of the table owner. You can omit owner names on the command line by specifying *dbname..tablename*. If multiple tables have the same name, and different owners, **optdiag** prints information for each table with that name.
Statistics for table	Name of the table.
Data page count	Number of data pages in the table.
Empty data page count	Count of pages that have deleted rows only.
Data row count	Number of data rows in the table.
Forwarded row count	Number of forwarded rows in the table. This value is always 0 for an allpages-locked table.
Deleted row count	Number of rows that have been deleted from the table. These are committed deletes where the space has not been reclaimed by one of the functions that clears deleted rows. This value is always 0 for an allpages-locked table.
Data page CR count	A counter used to derive the data page cluster ratio. See “Data Page CR Count” on page 19-6.

Table 19-2: Table statistics (continued)

Row Label	Information Provided
OAM + allocation page count	Number of OAM pages for the table, plus the number of allocation units in which the table occupies space. These statistics are used to estimate the cost of OAM scans on data-only-locked tables. The value is maintained only on data-only-locked tables.
First extent data pages	Number of pages that share the first extent in an allocation unit with the allocation page. These pages need to be read using 2K I/O, rather than large I/O. This information is maintained only for data-only-locked tables.
Data row size	Average length of a data row, in bytes. The size includes row overhead. This value is updated only by **update statistics**, **create index**, and **alter table...lock**.
Index height	Height of the index, not counting the leaf level. This row is included in the table-level output only for clustered indexes on allpages-locked tables. For all other indexes, the index height appears in the index-level output. This value does not apply to heap tables.

Data Page CR Count

The "Data Page CR count" is used to compute the data page cluster ratio, which can help determine the effectiveness of large I/O for table scans and range scans. This value is updated only when you run **update statistics**.

Table-Level Derived Statistics

The "Derived statistics" in the table-level section reports the statistics derived from the "Data Page CR count" and data page count. Table 19-3 describes the output.

Table 19-3: Cluster ratio for a table

Row Label	Information Provided
Data page cluster ratio	The data page cluster ratio is used to estimate the effectiveness of large I/O. It is used to estimate the number of I/Os required to read an allpages-locked table by following the page chain, and to estimate the number of large I/Os required to scan a data-only-locked table using an OAM scan.

Table 19-3: Cluster ratio for a table

Row Label	Information Provided
Space utilization	The ratio of the minimum space usage for this table, and the current space usage.
Large I/O efficiency	Estimates the number of useful pages brought in by each large I/O.

Data Page Cluster Ratio

For allpages-locked tables, the data page cluster ratio measures how well the pages are sequenced on extents, when the table is read in page-chain order. A cluster ratio of 1.0 indicates perfect sequencing. A lower cluster ratio indicates that the page chain is fragmented.

For data-only-locked tables, the data page cluster ratio measures how well the pages are packed on the extents. A cluster ratio of 1.0 indicates complete packing of extents. A low data page cluster ratio indicates that extents allocated to the table contain empty pages.

For an example of how the data page cluster ratio is used, see "How Cluster Ratios Affect Large I/O Estimates" on page 6-7.

Space Utilization

Space utilization uses the average row size and number of rows to compute the expected minimum number of data pages, and compares it to the current number of pages. If space utilization is low, running **reorg rebuild** on the table or dropping and re-creating the clustered index can reduce the amount of empty space on data pages, and the number of empty pages in extents allocated to the table. If you are using space management properties such as **fillfactor** or **reservepagegap**, the empty space that is left for additional rows on data pages of a table with a clustered index and the number of empty pages left in extents for the table affects the space utilization value.

If statistics have not been updated recently and the average row size has changed or the number of rows and pages are inaccurate, space utilization may report values greater than 1.0.

Large I/O Efficiency

Large I/O efficiency estimates the number of useful pages brought in by each large I/O. For examples, if the value is .5, a 16K I/O returns, on average, 4 2K pages needed for the query, and 4 other pages, either empty pages or pages that share the extent due to lack of clustering. Low values are an indication that re-creating the clustered index or running **reorg rebuild** on the table could improve I/O performance.

Index Statistics

This **optdiag** section is printed for each nonclustered index and for a clustered index on a data-only-locked table. Information for clustered indexes on allpages-locked tables is reported as part of the table statistics. Table 19-4 describes the output.

Sample Output for Index Statistics

```
Statistics for index:                   "title_id_ix"
(nonclustered)
Index column list:                      "title_id"
     Leaf count:                        45
     Empty leaf page count:             0
     Data page CR count:                4952.0000000000000000
     Index page CR count:               6.0000000000000000
     Data row CR count:                 4989.0000000000000000
     First extent leaf pages:           0
     Leaf row size:                     17.8905999999999992
     Index height:                      1

  Derived statistics:
     Data page cluster ratio:           0.0075819672131148
     Index page cluster ratio:          1.0000000000000000
     Data row cluster ratio:            0.0026634382566586
```

Table 19-4: Index statistics

Row Label	Information Provided
Statistics for index	Index name and type.
Index column list	List of columns in the index.
Leaf count	Number of leaf-level pages in the index.
Empty leaf page count	Number of empty leaf pages in the index.
Data page CR count	A counter used to compute the data page cluster r.atio for accessing a table using the index. See “Index-Level Derived Statistics” on page 19-9.

Table 19-4: Index statistics (continued)

Row Label	Information Provided
Index page CR count	A counter used to compute the index page cluster ratio. See "Index-Level Derived Statistics" on page 19-9.
Data row CR count	A counter used to compute the data row cluster ratio See "Index-Level Derived Statistics" on page 19-9.
First extent leaf pages	The number of leaf pages in the index stored in the first extent in an allocation unit. These pages need to be read using 2K I/O, rather than large I/O. This information is maintained only for indexes on data-only-locked tables.
Leaf row size	Average size of a leaf-level row in the index. This value is only updated by **update statistics**, **create index**, and **alter table...lock**.
Index height	Index height, not including the leaf level.

Index-Level Derived Statistics

The derived statistics in the index-level section are based on the "CR count" values shown in "Index Statistics" on page 19-8. Table 19-5 describes the output.

Table 19-5: Cluster ratios for a nonclustered index

Row Label	Information Provided
Data page cluster ratio	The fraction of row accesses that do not require an additional extent I/O because of storage fragmentation, while accessing rows in order by this index using large I/O. It is a measure of the sequencing of data pages on extents.
Index page cluster ratio	The fraction of index leaf page accesses via the page chain that do not require extra extent I/O. It is a measure of the sequencing of index pages on extents.
Data row cluster ratio	The fraction of data page accesses that do not require an extra I/O when accessing data rows in order by this index. It is a measure of the sequencing of rows on data pages.
Space utilization	The ratio of the minimum space usage for the leaf level of this index, and the current space usage.
Large I/O efficiency	Estimates the number of useful pages brought in by each large I/O.

Data Page Cluster Ratio

The data page cluster ratio is used to compute the effectiveness of large I/O when this index is used to access the data pages. If the table is perfectly clustered with respect to the index, the cluster ratio is 1.0. Data page cluster ratios can vary widely. They are often high for some indexes, and very low for others. See "How Cluster Ratios Affect Large I/O Estimates" on page 6-7 for more information.

Index Page Cluster Ratio

The index page cluster ratio is used to estimate the cost of large I/O for queries that need to read a large number of leaf-level pages from nonclustered indexes or clustered indexes on data-only-locked tables. Some examples of such queries are covered index scans and range queries that read a large number of rows.

On newly created indexes, the "Index page cluster ratio" is 1.0, or very close to 1.0, indicating optimal clustering of index leaf pages on extents. As index pages are split and new pages are allocated from additional extents, the ratio drops. A very low percentage could indicate that dropping and re-creating the index or running **reorg rebuild** on the index would improve performance, especially if many queries perform covered scans. See "How Cluster Ratios Affect Large I/O Estimates" on page 6-7 for more information.

Data Row Cluster Ratio

The data row cluster ratio is used to estimate the number of pages that need to be read while using this index to access the data pages. This ratio may be very high for some indexes, and quite low for others.

For an example of how the data row cluster ratio and the index page cluster ratio are used to optimize queries, see "How Data Row Cluster Ratio Refines Page Estimates" on page 6-13.

Space Utilization for an Index

Space utilization uses the average row size and number of rows to compute the expected minimum size of leaf-level index pages and compares it to the current number of leaf pages. If space utilization is low, running **reorg rebuild** on index or dropping and re-creating the index can reduce the amount of empty space on index pages, and the number of empty pages in extents allocated to the index. If you are using space management properties such as **fillfactor** or **reservepagegap**, the empty space that is left for additional rows on leaf pages, and the number of empty pages left in extents for the index affects space utilization.

If statistics have not been updated recently and the average row size has changed or the number of rows and pages are inaccurate, space utilization may report values greater than 1.0.

Large I/O Efficiency for an Index

Large I/O efficiency estimates the number of useful pages brought in by each large I/O. For examples, if the value is .5, a 16K I/O returns, on average, 4 2K pages needed for the query, and 4 other pages, either empty pages or pages that share the extent due to lack of clustering. Low values are an indication that re-creating indexes or running **reorg rebuild** could improve I/O performance.

Column Statistics

optdiag column-level statistics include:

- Statistics giving the density and selectivity of columns. If an index includes more than one column, **optdiag** prints the information described in Table 19-6 for each prefix subset of the index keys. If statistics are created using **update statistics** with a column name list, density statistics are stored for each prefix subset in the column list.
- A histogram, if the table contains one or more rows of data at the time the index is created or **update statistics** is run. There is a histogram for the leading column for:
  - Each index that currently exists (if there was at least one non-null value in the column when the index was created)
  - Any indexes that have been created and dropped (as long as **delete statistics** has not been run)
  - Any column list on which **update statistics** has been run

  There is also a histogram for:
  - Every column in an index, if the **update index statistics** command was used
  - Every column in the table, if the **update all statistics** command was used

optdiag also prints a list of the columns in the table for which there are no statistics. For example, here is a list of the columns in the *authors* table that do not have statistics:

```
No statistics for column(s):        "address"
(default values used)               "au_fname"
                                    "phone"
                                    "state"
                                    "zipcode"
```

Sample Output for Column Statistics

The following sample shows the statistics for the *city* column in the *authors* table:

```
Statistics for column:                  "city"
Last update of column statistics:       Jul 20 1998  6:05:26:656PM

     Range cell density:                0.0007283200000000
     Total density:                     0.0007283200000000
     Range selectivity:                 default used (0.33)
     In between selectivity:            default used (0.25)
```

Table 19-6 describes the output.

Table 19-6: Column statistics

Row Label	Information Provided
Statistics for column	Name of the column; if this block of information provides information about a prefix subset in a compound index or column list, the row label is “Statistics for column group.”
Last update of column statistics	Date the index was created, date that **update statistics** was last run, or date that **optdiag** was last used to change statistics.
Statistics originated from upgrade of distribution page	Statistics resulted from an upgrade of a pre-11.9 distribution page. This message is not printed if **update statistics** has been run on the table or index or if the index has been dropped and re-created after an upgrade. If this message appears in **optdiag** output, running **update statistics** is recommended.

Table 19-6: Column statistics (continued)

Row Label	Information Provided
Statistics loaded from Optdiag	**optdiag** was used to change *sysstatistics* information. **create index** commands print warning messages indicating that edited statistics are being overwritten. This row is not displayed if the statistics were generated by **update statistics** or **create index**.
Range cell density	Density for equality search arguments on the column. See "Range Cell and Total Density Values" on page 19-13.
Total density	Join density for the column. This value is used to estimate the number of rows that will be returned for a join on this column. See "Range Cell and Total Density Values" on page 19-13.
Range selectivity	Prints the default value of .33, unless the value has been updated using **optdiag** input mode. This is the value used for range queries if the search argument is not known at optimize time.
In between selectivity	Prints the default value of .25, unless the value has been updated using **optdiag** input mode. This is the value used for range queries if the search argument is not known at optimize time.

Range Cell and Total Density Values

Adaptive Server stores two values for the density of column values:

- The "Range cell density" measures the duplicate values only for range cells. If there are any frequency cells for the column, they are eliminated from the computation for the range-cell density. If there are only frequency cells for the column, and no range cells, the range-cell density is 0. See "Understanding Histogram Output" on page 19-17 for information on range and frequency cells.
- The "Total density" measures the duplicate values for all columns, those represented by both range cells and frequency cells.

Using two separate values improves the optimizer's estimates of the number of rows to be returned:

- If a search argument matches the value of a frequency cell, the fraction of rows represented by the weight of the frequency cell will be returned.

- If a search argument falls within a range cell, the range-cell density and the weight of the range cell are used to estimate the number of rows to be returned.

For joins, the optimizer bases its estimates on the average number of rows to be returned for each scan of the table, so the total density, which measures the average number of duplicates for all values in the column, provides the best estimate. The total density is also used for equality arguments when the value of the search argument is not known when the query is optimized. See "Range and In-Between Selectivity Values" on page 19-15 for more information.

For indexes on multiple columns, the range-cell density and total density are stored for each prefix subset. In the sample output below for an index on *titles (pub_id, type, pubdate)*, the density values decrease with each additional column considered.

```
Statistics for column:              "pub_id"
Last update of column statistics:   Feb  4 1998 12:58PM

     Range cell density:            0.0335391029690461
     Total density:                 0.0335470400000000

Statistics for column group:        "pub_id", "type"
Last update of column statistics:   Feb  4 1998 12:58PM

     Range cell density:            0.0039044009265108
     Total density:                 0.0039048000000000

Statistics for column group:        "pub_id", "type", "pubdate"
Last update of column statistics:   Feb  4 1998 12:58PM

     Range cell density:            0.0002011791956201
     Total density:                 0.0002011200000000
```

With 5000 rows in the table, the increasing precision of the optimizer's estimates of rows to be returned depends on the number of search arguments used in the query:

- An equality search argument on only *pub_id* results in the estimate that 0.0335391029690461 * 5000 rows, or 168 rows, will be returned.
- Equality search arguments for all three columns result in the estimate that 0.0002011791956201 * 5000 rows, or only 1 row will be returned.

This increasing level of accuracy as more search arguments are evaluated can greatly improve the optimization of many queries.

Range and In-Between Selectivity Values

optdiag prints the default values for range and in-between selectivity, or the values that have been set for these selectivities in an earlier **optdiag** session. These values are used for range queries when search arguments are not known when the query is optimized. For equality search arguments whose value is not known, the total density is used as the default.

Search arguments cannot be known at optimization time for:

- Stored procedures that set variables within a procedure
- Queries in batches that set variables for search arguments within a batch

Table 5-2 on page 5-17 shows the default values that are used. These approximations can result in suboptimal query plans because they either overestimate or underestimate the number of rows to be returned by a query. See "Updating Selectivities with optdiag Input Mode" on page 19-23 for information on using **optdiag** to supply selectivity values.

Histogram Displays

Histograms store information about the distribution of values in a column. Table 19-7 shows the commands that create and update histograms and which columns are affected.

Table 19-7: Commands that create histograms

Command	Histogram For
create index	Leading column only
update statistics	
table_name or *index_name*	Leading column only
column_list	Leading column only
update index statistics	All indexed columns
update all statistics	All columns

Sample Output for Histograms

```
Histogram for column:               "city"
Column datatype:                    varchar(20)
Requested step count:               20
Actual step count:                  20
```

optdiag first prints summary data about the histogram, as shown in Table 19-8.

Table 19-8: Histogram summary statistics

Row Label	Information Provided
Histogram for column	Name of the column.
Column datatype	Datatype of the column, including the length, precision and scale, if appropriate for the datatype.
Requested step count	Number of steps requested for the column.
Actual step count	Number of steps generated for the column. This number can be less than the requested number of steps if the number of distinct values in the column is smaller than the requested number of steps.

Histogram output is printed in columns, as described in Table 19-9.

Table 19-9: Columns in optdiag histogram output

Column	Information Provided
Step	Number of the step.
Weight	Weight of the step.
(Operator)	**<**, **<=**, or **=**, indicating the limit of the value. Operators differ, depending on whether the cell represents a range cell or a frequency call.
Value	Upper boundary of the values represented by a range cell or the value represented by a frequency count.

No heading is printed for the Operator column.

Understanding Histogram Output

A histogram is a set of cells in which each cell has a weight. Each cell has an upper bound and a lower bound, which are distinct values from the column. The weight of the cell is a floating-point value between 0 and 1, representing either:

- The fraction of rows in the table within the range of values, if the operator is **<=**, or
- The number of values that match the step, if the operator is **=**.

The optimizer uses the combination of ranges, weights, and density values to estimate the number of rows in the table that are to be returned for a query clause on the column.

Adaptive Server uses equi-height histograms, where the number of rows represented by each cell is approximately equal. For example, the following histogram on the *city* column on *pubtune..authors* has 20 steps; each step in the histogram represents about 5 percent of the table:

```
Step     Weight                  Value

   1     0.00000000     <=      "APO Miamh\377\377\377\377\377\377\377"
   2     0.05460000     <=      "Atlanta"
   3     0.05280000     <=      "Boston"
   4     0.05400000     <=      "Charlotte"
   5     0.05260000     <=      "Crown"
   6     0.05260000     <=      "Eddy"
   7     0.05260000     <=      "Fort Dodge"
   8     0.05260000     <=      "Groveton"
   9     0.05340000     <=      "Hyattsville"
  10     0.05260000     <=      "Kunkle"
  11     0.05260000     <=      "Luthersburg"
  12     0.05340000     <=      "Milwaukee"
  13     0.05260000     <=      "Newbern"
  14     0.05260000     <=      "Park Hill"
  15     0.05260000     <=      "Quicksburg"
  16     0.05260000     <=      "Saint David"
  17     0.05260000     <=      "Solana Beach"
  18     0.05260000     <=      "Thornwood"
  19     0.05260000     <=      "Washington"
  20     0.04800000     <=      "Zumbrota"
```

The first step in a histogram represents the proportion of null values in the table. Since there are no null values for *city*, the weight is 0. The value for the step that represents null values is represented by the highest value that is less than the minimum column value. For character strings, the value for the first cell is the highest possible string value less than the minimum column value (“APO Miami” in this example), padded to the defined column length with the highest character in the character set used by the server. What you actually see in your output depends on the character set, type of terminal, and software that you are using to view **optdiag** output files.

In the preceding histogram, the value represented by each cell includes the upper bound, but excludes the lower bound. The cells in this histogram are called **range cells**, because each cell represents a range of values.

The range of values included in a range cell can be represented as follows:

```
lower_bound < (values for cell) <= upper bound
```

In **optdiag** output, the lower bound is the value of the previous step, and the upper bound is the value of the current step. For example, in the histogram above, step 4 includes Charlotte (the upper bound), but excludes Boston (the lower bound). The weight for this step is .0526, indicating that 5.26 percent of the table matches the query clause:

```
where city > Boston and city <= "Charlotte"
```

The operator column in the **optdiag** histogram output shows the **<=** operator. Different operators are used for histograms with highly duplicated values.

Histograms for Columns with Highly Duplicated Values

Histograms for columns with highly duplicated values look very different from histograms for columns with a large number of discrete values. In histograms for columns with highly duplicated values, a single cell, called a **frequency cell**, represents the duplicated value. The weight of the frequency cell shows the percentage of columns that have matching values.

Histogram output for frequency cells varies, depending on whether the column values represent one of the following:

- A dense frequency count, where values in the column are contiguous in the domain. For example, 1, 2, 3 are contiguous integer values
- A sparse frequency count, where the domain of possible values contains values not represented by the discrete set of values in the table
- A mix of dense and sparse frequency counts.

Histogram output for some columns includes a mix of frequency cells and range cells.

Histograms for Dense Frequency Counts

The following output shows the histogram for a column that has 6 distinct integer values, 1–6, and some null values:

```
Step     Weight                Value

   1     0.13043478      <     1
   2     0.04347826      =     1
   3     0.17391305     <=     2
   4     0.30434781     <=     3
   5     0.13043478     <=     4
   6     0.17391305     <=     5
   7     0.04347826     <=     6
```

The histogram above shows a **dense frequency count**, because all the values for the column are contiguous integers. The first cell represents null values. Since there are null values, the weight for this cell represents the percentage of null values in the column. The "Value" column for the first step displays the minimum column value in the table and the **<** operator.

Histograms for Sparse Frequency Counts

In a histogram representing a column with a **sparse frequency count**, the highly duplicated values are represented by a step showing the discrete values with the **=** operator and the weight for the cell. Preceding each step, there is a step with a weight of 0.0, the same value, and the **<** operator, indicating that there are no rows in the table with intervening values. For columns with null values, the first step will have a nonzero weight if there are null values in the table.

The following histogram represents the *type* column of the *titles* table. Since there are only 9 distinct types, they are represented by 18 steps.

```
Step     Weight                    Value

    1     0.00000000        <       "UNDECIDED   "
    2     0.11500000        =       "UNDECIDED   "
    3     0.00000000        <       "adventure   "
    4     0.11000000        =       "adventure   "
    5     0.00000000        <       "business    "
    6     0.11040000        =       "business    "
    7     0.00000000        <       "computer    "
    8     0.11640000        =       "computer    "
```

```
 9     0.00000000        <       "cooking     "
10     0.11080000        =       "cooking     "
11     0.00000000        <       "news        "
12     0.10660000        =       "news        "
13     0.00000000        <       "psychology  "
14     0.11180000        =       "psychology  "
15     0.00000000        <       "romance     "
16     0.10800000        =       "romance     "
17     0.00000000        <       "travel      "
18     0.11100000        =       "travel      "
```

For example, 10.66% of the values in the type column are "news," so for a table with 5000 rows, the optimizer estimates that 533 rows will be returned.

Histograms for Columns with Sparse and Dense Values

For tables with some values that are highly duplicated, and others that have distributed values, the histogram output shows a combination of operators and a mix of frequency cells and range cells. The column represented in the histogram below has a value of 30.0 for a large percentage of rows, a value of 50.0 for a large percentage of rows, and a value 100.0 for another large percentage of rows. There are two steps in the histogram for each of these values: one step representing the highly duplicated value has the **=** operator and a weight showing the percentage of columns that match the value. The other step for each highly duplicated value has the **<** operator and a weight of 0.0. The datatype for this column is *numeric(5,1)*.

```
Step     Weight                Value

   1     0.00000000     <=     0.9
   2     0.04456094     <=     20.0
   3     0.00000000      <     30.0
   4     0.29488859      =     30.0
   5     0.05996068     <=     37.0
   6     0.04292267     <=     49.0
   7     0.00000000      <     50.0
   8     0.19659241      =     50.0
   9     0.06028834     <=     75.0
  10     0.05570118     <=     95.0
  11     0.01572739     <=     99.0
  12     0.00000000      <     100.0
  13     0.22935779      =     100.0
```

Since the lowest value in the column is 1.0, the step for the null values is represented by 0.9.

Choosing the Number of Steps for Highly Duplicated Values

The histogram examples for frequency cells in this section use a relatively small number of highly duplicated values, so the resulting histograms require less than 20 steps, which is the default number of steps for **create index** or **update statistics**. If your table contains a large number of highly duplicated values for a column, and the distribution of keys in the column is not uniform, increasing the number of steps in the histogram can allow the optimizer to produce more accurate cost estimates for queries with search arguments on the column.

For columns with dense frequency counts, the number of steps should be at least one greater than the number of values, to allow a step for the cell representing null values.

For columns with sparse frequency counts, use at least twice as many steps as there are distinct values. This allows for the intervening cells with zero weights, plus the cell to represent the null value. For example, if the *titles* table in the *pubtune* database has 30 distinct prices, this **update statistics** command creates a histogram with 60 steps:

```
update statistics titles
using 60 values
```

This **create index** command specifies 60 steps:

```
create index price_ix on titles(price)
with statistics using 60 values
```

If a column contains some values that match very few rows, these may still be represented as range cells, and the resulting number of histogram steps will be smaller than the requested number. For example, requesting 100 steps for a *state* column may generate some range cells for those states represented by a small percentage of the number of rows.

Changing Statistics with *optdiag*

A System Administrator can use **optdiag** to change column-level statistics.

◆ ***WARNING!***

Using optdiag to alter statistics can improve the performance of some queries. Remember, however, that optdiag overwrites existing information in the system tables, which can affect all queries for a given table. Use extreme caution and test all changes thoroughly on all queries that use the table. If possible, test the changes using optdiag simulate on a development server before loading the statistics into a production server. If you load statistics without simulate mode, be prepared to restore the statistics, if necessary, either by using an untouched copy of optdiag output or by rerunning update statistics.

Do not attempt to change any statistics by running an update, delete, or insert command.

After you change statistics using **optdiag**, running **create index** or **update statistics** overwrites the changes. The commands succeed, but print a warning message. This message indicates that altered statistics for the *titles.type* column have been overwritten:

```
WARNING: Edited statistics are overwritten. Table:
'titles' (objectid 208003772), column: 'type'.
```

Using the *optdiag* Binary Mode

Because precision can be lost with floating point numbers, **optdiag** provides a binary mode. The following command displays both human-readable and binary statistics:

```
optdiag binary statistics pubtune..titles.price
        -Usa -Ppasswd -o price.opt
```

In binary mode, any statistics that can be edited with **optdiag** are printed twice, once with binary values, and once with floating-point values. The lines displaying the float values start with the **optdiag** comment character, the pound sign (#). This sample shows the first few rows of the histogram for the *city* column in the *authors* table:

```
Step  Weight            Value

  1   0x3d2810ce    <=    0x41504f204d69616d68ffffffffffffffffffffff
# 1   0.04103165    <=    "APO Miamh\377\377\377\377\377\377\377\377"
  2   0x3d5748ba    <=    0x41746c616e7461
# 2   0.05255959    <=    "Atlanta"
  3   0x3d5748ba    <=    0x426f79657273
# 3   0.05255959    <=    "Boyers"
  4   0x3d58e27d    <=    0x4368617474616e6f6f6761
# 4   0.05295037    <=    "Chattanooga"
```

When **optdiag** loads this file, all uncommented lines are read, while all characters following the pound sign are ignored. To edit the float values instead of the binary values, remove the pound sign from the lines displaying the float values, and insert the pound sign at the beginning of the corresponding line displaying the binary value.

When You Must Use Binary Mode

Two histogram steps in **optdiag** output can show the same value due to loss of precision, even though the binary values differ. For example, both 1.999999999 and 2.000000000 may be displayed as 2.000000000 in decimal, even though the binary values are 0x3fffffffffbb47d0 and 0x4000000000000000. In these cases, you should use binary mode for input.

If you do not use binary mode, **optdiag** issues an error message indicating that the step values are not increasing and telling you to use binary mode. **optdiag** skips loading the histogram in which the error occurred, to avoid losing precision in *sysstatistics*.

Updating Selectivities with *optdiag* Input Mode

You can use **optdiag** to customize the server-wide default values for selectivities to match the data for specific columns in your application. The optimizer uses range and in-between selectivity values when the value of a search argument is not known when a query is optimized. The server-wide defaults are:

- Range selectivity – 0.33
- In-between selectivity – 0.25

You can use **optdiag** to provide values to be used to optimize queries on a specific column. The following example shows how **optdiag** displays default values:

```
Statistics for column:              "city"
Last update of column statistics:   Feb  4 1998  8:42PM

     Range cell density:            0x3f634d23b702f715
#    Range cell density:            0.0023561189228464
     Total density:                 0x3f46fae98583763d
#    Total density:                 0.0007012977830773
     Range selectivity:             default used (0.33)
#    Range selectivity:             default used (0.33)
     In between selectivity:        default used (0.25)
#    In between selectivity:        default used (0.25)
```

To edit these values, replace the entire “default used (0.33)” or “default used (0.25)” string with a float value. The following example changes the range selectivity to .25 and the in-between selectivity to .05, using decimal values:

```
     Range selectivity:             0.200000000
     In between selectivity:        0.050000000
```

Editing Histograms

You can edit histograms to:

- Remove a step, by transferring its weight to an adjacent line and deleting the step
- Add a step or steps, by spreading the weight of a cell to additional lines, with the upper bound for column values the step is to represent

Adding Frequency Count Cells to a Histogram

One common reason for editing histograms is to add frequency count cells without greatly increasing the number of steps. The changes you will need to make to histograms vary, depending on whether the values represent a dense or sparse frequency count.

Editing a Histogram with a Dense Frequency Count

If the next lesser column value to the step to be changed is as close as possible to the frequency count value, then the frequency count cell can be extracted simply.

For example, if a column contains at least one 19 and many 20s, and the histogram uses a single cell to represent all the values greater than 17 and less than or equal to 22, **optdiag** output shows the following information for the cell:

```
Step     Weight            Value
...
4     0.100000000    <=     17
5     0.400000000    <=     22
...
```

Altering this histogram to place the value 20 on its own step requires adding two steps, as shown here:

```
...
4     0.100000000    <=     17
5     0.050000000    <=     19
6     0.300000000    <=     20
7     0.050000000    <=     22
...
```

In the altered histogram above, step 5 represents all values greater than 17 and less than or equal to 19. The sum of the weights of steps 5, 6, and 7 in the modified histogram equals the original weight value for step 5.

Editing a Histogram with a Sparse Frequency Count

If the column has no values greater than 17 and less than 20, the representation for a sparse frequency count must be used instead. Here are the original histogram steps:

```
Step     Weight            Value
...
4     0.100000000    <=     17
5     0.400000000    <=     22
...
```

The following example shows the zero-weight step, step 5, required for a sparse frequency count:

```
...
4     0.100000000    <=     17
5     0.000000000     <     20
6     0.350000000     =     20
7     0.050000000    <=     22
...
```

The operator for step 5 must be **<**. Step 6 must specify the weight for the value 20, and its operator must be **=**.

Skipping the Load-Time Verification for Step Numbering

By default, **optdiag** input mode checks that the numbering of steps in a histogram increases by 1. To skip this check after editing histogram steps, use the command line flag **-T4**:

```
optdiag statistics pubtune..titles -Usa -Ppassword
    -T4 -i titles.opt
```

Rules Checked During Histogram Loading

During histogram input, the following rules are checked, and error messages are printed if the rules are violated:

- The step numbers must increase monotonically, unless the **-T4** command line flag is used.
- The column values for the steps must increase monotonically.
- The weight for each cell must be between 0.0 and 1.0.
- The total of weights for a column must be close to 1.0.
- The first cell represents null values and it must be present, even for columns that do not allow null values. There must be only one cell representing the null value.
- Two adjacent cells cannot both use the **<** operator.

Re-Creating Indexes Without Losing Statistics Updates

If you need to drop and re-create an index after you have updated a histogram, and you want to keep the edited values, specify 0 for the number of steps in the **create index** command. This command re-creates the index without changing the histogram:

```
create index title_id_ix on titles(title_id)
with statistics using 0 values
```

Using Simulated Statistics

optdiag can generate statistics that can be used to simulate a user environment without requiring a copy of the table data. This permits analysis of query optimization using a very small database. For example, simulated statistics can be used:

- For Technical Support replication of optimizer problems
- To perform “what if” analysis to plan configuration changes

In most cases, you will use simulated statistics to provide information to Technical Support or to perform diagnostics on a development server. See "Requirements for Loading and Using Simulated Statistics" on page 19-29 for information on setting up a separate database for using simulated statistics.

You can also load simulated statistics into the database from which they were copied. Simulated statistics are loaded into the system tables with IDs that distinguish them from the actual table data. The **set statistics simulate on** command instructs the server to optimize queries using the simulated statistics, rather than the actual statistics.

optdiag Syntax for Simulated Statistics

This command displays simulate-mode statistics for the *pubtune* database:

```
optdiag simulate statistics pubtune -o pubtune.sim
```

If you want binary simulated output, use:

```
optdiag binary simulate statistics pubtune -o pubtune.sim
```

To load these statistics, use:

```
optdiag simulate statistics -i pubtune.sim
```

Simulated Statistics Output

Output for the **simulate** option to **optdiag** prints a row labeled "simulated" for each row of statistics, except histograms. You can modify and load the simulated values, while retaining the file as a record of the actual values.

- If **binary** mode is specified, there are three rows of output:
  - A binary "simulated" row
  - A decimal "simulated" row, commented out
  - A decimal "actual" row, commented out
- If **binary** mode is not specified, there are two rows:
  - A "simulated" row
  - An "actual" row, commented out

Here is a sample of the table-level statistics for the *titles* table in the *pubtune* database:

```
Table owner:                        "dbo"
Table name:                         "titles"

Statistics for table:               "titles"

     Data page count:               731.0000000000000000 (simulated)
#    Data page count:               731.0000000000000000 (actual)
     Empty data page count:         1.0000000000000000 (simulated)
#    Empty data page count:         1.0000000000000000 (actual)
     Data row count:                5000.0000000000000000 (simulated)
#    Data row count:                5000.0000000000000000 (actual)
     Forwarded row count:           0.0000000000000000 (simulated)
#    Forwarded row count:           0.0000000000000000 (actual)
     Deleted row count:             0.0000000000000000 (simulated)
#    Deleted row count:             0.0000000000000000 (actual)
     Data page CR count:            0.0000000000000000 (simulated)
#    Data page CR count:            0.0000000000000000 (actual)
     OAM + allocation page count:   6.0000000000000000 (simulated)
#    OAM + allocation page count:   6.0000000000000000 (actual)
     First extent data pages:       0.0000000000000000 (simulated)
#    First extent data pages:       0.0000000000000000 (actual)
     Data row size:                 190.0000000000000000 (simulated)
#    Data row size:                 190.0000000000000000 (actual)
```

In addition to table and index statistics, the **simulate** option to **optdiag** copies out:

- Partitioning information for partitioned tables. If a table is partitioned, these two lines appear at the end of the table statistics:

```
     Pages in largest partition:    390.0000000000000000 (simulated)
#    Pages in largest partition:    390.0000000000000000 (actual)
```

- Settings for the parallel processing configuration parameters:

```
Configuration Parameters:
     Number of worker processes:     20 (simulated)
#    Number of worker processes:     20 (actual)
     Max parallel degree:           10 (simulated)
#    Max parallel degree:           10 (actual)
     Max scan parallel degree:      3 (simulated)
#    Max scan parallel degree:      3 (actual)
```

- Cache configuration information for the default data cache and the caches used by the specified database or the specified table and its indexes. If *tempdb* is bound to a cache, that cache’s configuration is also included. Here is sample output for the cache used by the *pubtune* database:

```
Configuration for cache:               "pubtune_cache"

     Size of 2K pool in Kb:            15360 (simulated)
#    Size of 2K pool in Kb:            15360 (actual)
     Size of 4K pool in Kb:            0 (simulated)
#    Size of 4K pool in Kb:            0 (actual)
     Size of 8K pool in Kb:            0 (simulated)
#    Size of 8K pool in Kb:            0 (actual)
     Size of 16K pool in Kb:           0 (simulated)
#    Size of 16K pool in Kb:           0 (actual)
```

If you want to test how queries use a 16K pool, you could alter the simulated statistics values above to read:

```
Configuration for cache:               "pubtune_cache"

     Size of 2K pool in Kb:            10240 (simulated)
#    Size of 2K pool in Kb:            15360 (actual)
     Size of 4K pool in Kb:            0 (simulated)
#    Size of 4K pool in Kb:            0 (actual)
     Size of 8K pool in Kb:            0 (simulated)
#    Size of 8K pool in Kb:            0 (actual)
     Size of 16K pool in Kb:           5120 (simulated)
#    Size of 16K pool in Kb:           0 (actual)
```

Requirements for Loading and Using Simulated Statistics

To use simulated statistics, you must issue the **set statistics simulate on** command before running the query. For more information, see "Running Queries with Simulated Statistics" on page 19-31.

To accurately simulate queries:

- Use the same locking scheme and partitioning for tables
- Re-create any triggers that exist on the tables and use the same referential integrity constraints
- Set any nondefault cache strategies and any nondefault concurrency optimization values
- Bind databases and objects to the caches used in the environment you are simulating
- Include any **set** options that affect query optimization (such as **set parallel_degree**) in the batch you are testing
- Create any view used in the query
- Use cursors, if they are used for the query

- Use a stored procedure, if you are simulating a procedure execution

Simulated statistics can be loaded into the original database, or into a database created solely for performing “what-if” analysis on queries.

Using Simulated Statistics in the Original Database

When the statistics are loaded into the original database, they are placed in separate rows in the system tables, and do not overwrite existing non-simulated statistics. The simulated statistics are only used for sessions where the **set statistics simulate** command is in effect.

While simulated statistics are not used to optimize queries for other sessions, executing any queries by using simulated statistics may result in query plans that are not optimal for the actual tables and indexes, and executing these queries may adversely affect other queries on the system.

Using Simulated Statistics in Another Database

When statistics are loaded into a database created solely for performing “what-if” analysis on queries, the following steps must be performed first:

- The database named in the input file must exist; it can be as small as 2MB. Since the database name occurs only once in the input file, you can change the database name, for example, from *production* to *test_db*.
- All tables and indexes included in the input file must exist, but the tables do not need to contain data.
- All caches named in the input file must exist. They can be the smallest possible cache size, 512K, with only a 2K pool. The simulated statistics provide the information for pool configuration.

Dropping Simulated Statistics

Loading simulated statistics adds rows describing cache configuration to the *sysstatistics* table in the *master* database. To remove these statistics, use **delete shared statistics**. The command has no effect on the statistics in the database where the simulated statistics were loaded.

If you have loaded simulated statistics into a database that contains real table and index statistics, you can drop simulated statistics in one of these ways:

- Use **delete statistics** on the table which deletes all statistics, and run **update statistics** to re-create only the nonsimulated statistics.
- Use **optdiag** (without **simulate** mode) to copy statistics out; then run **delete statistics** on the table, and use **optdiag** (without **simulate** mode) to copy statistics in.

Running Queries with Simulated Statistics

set statistics simulate on tells the optimizer to optimize queries using simulated statistics:

```
set statistics simulate on
```

In most cases, you also want to use **set showplan on** or **dbcc traceon(302)**.

If you have loaded simulated statistics into a production database, use **set noexec on** when you run queries using simulated statistics so that the query does not execute based on statistics that do not match the actual tables and indexes. This lets you examine the output of **showplan** and **dbcc traceon(302)** without affecting the performance of the production system.

showplan Messages for Simulated Statistics

When **set statistics simulate** is enabled and there are simulated statistics available, **showplan** prints the following message:

```
Optimized using simulated statistics.
```

If the server on which the simulation tests are performed has the parallel query options set to smaller values than the simulated values, **showplan** output first displays the plan using the simulated statistics, and then an adjusted query plan. If **set noexec** is turned on, the adjusted plan is not displayed.

Character Data Containing Quotation Marks

In histograms for character and datetime columns, all column data is contained in double quotes. If the column itself contains the double-quote character, **optdiag** displays two quotation marks. If the column value is:

```
a quote " mark
```

optdiag displays:

```
"a quote "" mark"
```

The only other special character in **optdiag** output is the pound sign, (#). In input mode, all characters on the line following a pound sign are ignored, except when the pound sign occurs within quotation marks as part of a column name or column value.

Effects of SQL Commands on Statistics

The information stored in *systabstats* and *sysstatistics* is affected by data definition language (DDL). Some data modification language also affects *systabstats*. Table 19-10 summarizes how DDL affects the *systabstats* and *sysstatistics* tables.

Table 19-10:Effects of DDL on systabstats and sysstatistics

Command	Effect on *systabstats*	Effect on *sysstatistics*
alter table...lock	Changes values to reflect the changes to table and index structure and size. When changing from allpages locking to data-only locking, the *indid* for clustered indexes is set to 0 for the table, and a new row is inserted for the index.	Same as **create index**, if changing from allpages to data-only locking or vice versa; no effect on changing between data-only locking schemes.
alter table to add, drop or modify a column definition	If the change affects the length of the row so that copying the table is required,	
create table	Adds a row for the table. If a constraint creates an index, see the **create index** commands below.	No effect, unless a constraint creates an index. See the **create index** commands below.
create clustered index	For allpages-locked tables, changes *indid* to 1 and updates columns that are pertinent to the index; for data-only-locked tables, adds a new row.	Adds rows for columns not already included; updates rows for columns already included.
create nonclustered index	Adds a row for the nonclustered index.	Adds rows for columns not already included; updates rows for columns already included.

Table 19-10:Effects of DDL on systabstats and sysstatistics (continued)

Command	Effect on *systabstats*	Effect on *sysstatistics*
delete statistics	No effect.	Deletes all rows for a table or just the rows for a specified column.
drop index	Removes rows for nonclustered indexes and for clustered indexes on data-only-locked tables. For clustered indexes on allpages-locked tables, sets the *indid* to 0 and updates column values.	Does not delete actual statistics for the indexed columns. This allows the optimizer to continue to use this information. Deletes simulated statistics for nonclustered indexes. For clustered indexes on allpages-locked tables, changes the value for the index ID in the row that contains simulated table data.
drop table	Removes all rows for the table.	Removes all rows for the table.
reorg	Updates restart points, if used with a time limit; updates number of pages and cluster ratios if page counts change; affects other values such as empty pages, forwarded or deleted row counts, depending on the option used.	The **rebuild** option recreates indexes.
truncate table	Resets values to reflect an empty table. Some values, like row length, are retained.	No effect; this allows reloading a truncated table without rerunning **update statistics**.
update statistics		
table_name	Updates values for the table and for all indexes on the specified table.	Updates histograms for the leading column of each index on the table; updates the densities for all indexes and prefix subsets of indexes.
index_name	Updates values for the specified index.	Updates the histogram for the leading column of the specified index; updates the densities for the prefix subsets of the index.
column_name(s)	No effect.	Updates or creates a histogram for a column and updates or creates densities for the prefix subsets of the specified columns.
update index statistics		

Table 19-10:Effects of DDL on systabstats and sysstatistics (continued)

Command	Effect on *systabstats*	Effect on *sysstatistics*
table_name	Updates values for the table and for all columns in all indexes on the specified table.	Updates histograms for all columns of each index on the table; updates the densities for all indexes and prefix subsets of indexes.
index_name	Updates values for the specified index	Updates the histogram for all column of the specified index; updates the densities for the prefix subsets of the index.
update all statistics		
table_name	Updates values for the table and for all columns in the specified table.	Updates histograms for all columns on the table; updates the densities for all indexes and prefix subsets of indexes.

How Query Processing Affects *systabstats*

Data modification can affect many of the values in the *systabstats* table. To improve performance, these values are changed in memory and flushed to *systabstats* periodically by the housekeeper task.

If you need to query *systabstats* directly, you can flush the in-memory statistics to *systabstats* with **sp_flushstats**. This command flushes the statistics for the *titles* table and any indexes on the table:

```
sp_flushstats titles
```

If you do not provide a table name, **sp_flushstats** flushes statistics for all tables in the current database.

➤ ***Note***

Some statistics, particularly cluster ratios, may be slightly inaccurate because not all page allocations and deallocations are recorded during changes made by data modification queries. Run **update statistics** or **create index** to correct any inconsistencies.

20 Advanced Optimizing Tools

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

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 may 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 by including commands in a query batch or in the text of the query:

- 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 forced options 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 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:

```
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.

set forceplan forces only join order, and not join type. There is no command for specifying the join type; you can disable merge joins at the server or session level. See “Enabling and Disabling Merge Joins” on page 20-12 for more information.

Risks of Using *forceplan*

Forcing join order has these risks:

- Misuse can lead to extremely expensive queries. Always test the query thoroughly with **statistics io**, and with and without **forceplan**.
- 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*

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.
- Use **update statistics** to add statistics for search arguments on unindexed search clauses in the query, especially for search arguments that match minor keys in compound indexes.
- If the query joins more than four tables, use **set table count** to see if it results in an improved join order. See “Specifying the Number of Tables Considered by the Optimizer” on page 20-4.

Specifying the Number of Tables Considered by the Optimizer

Adaptive Server optimizes joins by considering permutations of two to four tables at a time, as described in “Costing and Optimizing Joins” on page 7-1. If you suspect that an inefficient join order is being chosen for a join query, 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
```

Valid values are 0 though 8; the value 0 restores the default behavior.

For example, to specify 4-at-a-time optimization, use:

```
set table count 4
```

dbcc traceon(310) reports the number of tables considered at a time. See “dbcc traceon(310) and Final Query Plan Costs” on page 18-18 for more information.

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.

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. Some examples are:

- If you think that a more optimal join order can shorten total query optimization and execution time, especially for stored procedures that you expect to be executed many times once a plan is in the procedure cache
- When saving abstract plans for later use

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

You can specify the index to use for a query using the **(index *index_name*)** clause in **select**, **update**, and **delete** statements. 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 [correlation_name]
       (index {index_name | table_name } )
       [, table_name ...]
   where ...

delete table_name
   from table_name [correlation_name]
   (index {index_name | table_name }) ...

update table_name set col_name = value
   from table_name [correlation_name]
   (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 io** 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 queries:
  - On tables that have skewed data distribution
  - Range queries, since the access methods for these queries are sensitive to the size of the range

Use this option only after testing to be certain that the query performs better with the specified index option. Once you include an index specification in a query, you should check regularly to be sure that the resulting plan is still better than 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 alternative access method.
- Maintenance increases, 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.
- If the index is a composite index, run **update statistics** on the minor keys in the index, if they are used as search arguments. This can greatly improve optimizer cost estimates. Creating statistics for other columns frequently used for search clauses can also improve estimates.

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 scan a large number of index leaf pages

If the cache used by the table or index is configured for 16K I/O, a single I/O can read up to eight pages simultaneously. 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 15, “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 performance 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
```

Index Type and Large I/O

When you specify an I/O size with prefetch, the specification can affect both the data pages and the leaf-level index pages. Table 20-1 shows the effects.

Table 20-1: Access methods and prefetching

Access Method	Large I/O Performed On
Table scan	Data pages
Clustered index	Data pages only, for allpages-locked tables Data pages and leaf-level index pages for data-only-locked tables
Nonclustered index	Data pages and leaf pages of nonclustered index

showplan reports the I/O size used for both data and leaf-level pages. See "I/O Size Messages" on page 17-41 for more information.

When *prefetch* Specification Is Not Followed

In most cases, 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 disable 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 39-71.

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:

```
set prefetch off
```

To reenable large I/O, use:

```
set prefetch on
```

If large I/O is turned off for an object using **sp_cachestrategy, 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 3-21 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 nested-loop join, if the inner table is larger than the cache
- The outer table of a nested-loop join, since it needs to be read only once
- Both tables in a merge join

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 20-7.

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 **sp_cachestrategy**. The syntax is:

```
sp_cachestrategy dbname , [ownername.]tablename
   [, indexname | "text only" | "table only"
   [, { prefetch | mru }, { "on" | "off"}]]
```

This command turns off the large I/O prefetch strategy 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**, with 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.

Enabling and Disabling Merge Joins

By default, merge joins are not enabled at the server level. When merge joins are disabled, the server only costs nested-loop joins, and merge joins are not considered. To enable merge joins server-wide, set **enable sort-merge joins and JTC** to 1. This also enables join transitive closure.

The command **set sort_merge on** overrides the server level to allow use of merge joins in a session or stored procedure.

To enable merge joins, use:

```
set sort_merge on
```

To disable merge joins, use:

```
set sort_merge off
```

For information on configuring merge-joins server-wide see **enable sort-merge joins and JTC** in the *System Administration Guide*.

Enabling and Disabling Join Transitive Closure

By default, join transitive closure is not enabled at the server-level, since it can increase optimization time. You can enable join transitive closure at a session level with **set jtc on**. The session-level command overrides the server-level setting for the **enable sort-merge joins and JTC** configuration parameter.

For queries that execute quickly, even when several tables are involved, join transitive closure may increase optimization time with little benefit of an improvement in execution cost. For example, with join transitive closure applied to this query, the number of possible joins is multiplied for each added table:

```
select * from t1, t2, t3, t4, ... tN
where t1.c1 = t2.c1
and t1.c1 = t3.c1
and t1.c1 = t4.c1
...
and t1.c1 = tN.c1
```

For joins on very large tables, however, the additional optimization time involved in costing the join orders added by join transitive closure may result in a join order that greatly reduces query execution cost, improving the response time.

You can use **set statistics time** to check how long it takes to optimize the query. If running queries with **set jtc on** greatly increases optimization time, but also improves query execution by choosing a better join order, check the **showplan** or **dbcc traceon(302, 310)** output. Explicitly add the useful join orders to the query text. You can run the query without join transitive closure, and get the improved execution time, without the increased optimization time of examining all possible join orders generated by join transitive closure.

You can also enable join transitive closure and save abstract plans for queries that benefit. If you then execute those queries with loading from the saved plans enabled, the saved execution plan is used to optimize the query, making optimization time extremely short. See Chapter 21, “Introduction to Abstract Plans,” for more information on using abstract plans.

For information on configuring join transitive closure server-wide see **enable sort-merge joins and JTC** in the *System Administration Guide*.

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 20-2 shows how to combine the **index** and **parallel** keywords to obtain serial or parallel scans.

Table 20-2: Optimizer hints for serial and parallel execution

To Specify This Type of Scan:	Use This Syntax:
Parallel partition scan	**(index *tablename* parallel *N*)**
Parallel index scan	**(index *index_name* parallel *N*)**
Serial table scan	**(index *tablename* parallel 1)**
Serial index scan	**(index *index_name* parallel 1)**
Parallel, with the choice of table or index scan left to the optimizer	**(parallel *N*)**
Serial, with the choice of table or index scan left to the optimizer	**(parallel 1)**

When you specify the parallel degree for a table in a merge join, it affects the degree of parallelism used for both the scan of the table and the merge join.

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 of an inner query block 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 "Access Methods and Costing for or and in Clauses" on page 6-23.)
- 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 index to be used in the query, and sets the degree of parallelism to 5:

```
select * from titles
    (index title_id_clix parallel 5)
where ...
```

To force a table scan, use the table name instead of the index name.

Concurrency Optimization for Small Tables

For data-only-locked tables of 15 pages or fewer, Adaptive Server does not consider a table scan if there is a useful index on the table. Instead, it always chooses the cheapest index that matches any optimizable search argument in the query. The locking required for an index scan provides higher concurrency and reduces the chance of deadlocks, although slightly more I/O may be required than for a table scan.

If concurrency on small tables is not an issue, and you want to optimize the I/O instead, you can disable this optimization with **sp_chgattribute**. This command turns off concurrency optimization for a table:

```
sp_chgattribute tiny_lookup_table,
    "concurrency_opt_threshold", 0
```

With concurrency optimization disabled, the optimizer can choose table scans when they require fewer I/Os.

You can also increase the concurrency optimization threshold for a table. This command sets the concurrency optimization threshold for a table to 30 pages:

```
sp_chgattribute lookup_table,
    "concurrency_opt_threshold", 30
```

The maximum value for the concurrency optimization threshold is 32,767. Setting the value to -1 enforces concurrency optimization for a table of any size. It may be useful in cases where a table scan is chosen over indexed access, and the resulting locking results in increased contention or deadlocks.

The current setting is stored in *systabstats.conopt_thld* and is printed as part of **optdiag** output.

Changing the Locking Scheme

Concurrency optimization affects only data-only-locked tables. Table 20-3 shows the effect of changing the locking scheme.

Table 20-3: Effects of alter table on concurrency optimization settings

Changing Locking Scheme from	Effect on Stored Value
Allpages to data-only	Set to 15, the default
Data-only to allpages	Set to 0
One data-only scheme to another	Configured value retained

Query Tuning With Abstract Plans

21 Introduction to Abstract Plans

This chapter provides an overview of abstract plans. It contains the following sections:

What Are Abstract Plans?

Adaptive Server can capture query text and save an abstract plan for a query in the *sysqueryplans* system table. Using a rapid hashing method, incoming SQL queries can be compared to stored query text, and if a match is found, the saved abstract plan is used to execute the query.

An abstract plan describes the execution plan for a query using a language created for that purpose. This language contains operators to specify the choices and actions that can be generated by the optimizer. For example, to specify an index scan on the *titles* table, using the index *title_id_ix*, the abstract plan says:

```
( i_scan title_id_ix titles)
```

Abstract plans provide a means for System Administrators and performance tuners to protect the overall performance of a server from changes to query plans. Changes in query plans can arise due to:

- Adaptive Server software upgrades that affect optimizer choices and query plans
- New Adaptive Server features that change query plans
- Changing tuning options such as the parallel degree, table partitioning, or indexing

The major purpose of abstract plans is to provide a means to capture query plans before and after major system changes. The sets of before-and-after query plans can be compared to determine the effects of changes on your queries. Other uses include:

- Searching for specific types of plans, such as table scans or reformatting
- Searching for plans that use particular indexes
- Specifying full or partial plans for poorly-performing queries
- Saving plans for queries with long optimization times

Abstract plans provide an alternative to options that must be specified in the batch or query in order to influence optimizer decisions. Using abstract plans, you can influence the optimization of a SQL statement without having to modify the statement syntax. While matching query text to stored text requires some processing overhead, using a saved plan reduces query optimization overhead.

The Relationship Between Query Text and Query Plans

For most SQL queries, there are many possible query execution plans. SQL describes the desired result set, but does not describe how that result set should be obtained from the database. Consider a query that joins three tables, such as this:

```
select t1.c11, t2.c21
from t1, t2, t3
where t1.c11 = t2.c21
and t1.c11 = t3.c31
```

There are many different possible join orders, and depending on the indexes that exist on the tables, many possible access methods, including table scans, index scans, and the reformatting strategy. Each join may use either a nested-loop join or a merge join. These choices are determined by the optimizer's query costing algorithms, and are not included in or specified in the query itself.

When you capture the abstract plan, the query is optimized in the usual way, except that the optimizer also generates an abstract plan, and saves the query text and abstract plan in *sysqueryplans*.

The Limits of Options for Influencing Query Plans

Adaptive Server provides other options for influencing optimizer choices:

- Session-level options such as **set forceplan** to force join order or **set parallel_degree** to specify the maximum number of worker processes to use for the query

- Options that can be included in the query text to influence the index choice, cache strategy, and parallel degree

There are some limitations to using **set** commands or adding hints to the query text:

- Not all query plan steps can be influenced, for example, subquery attachment
- Some query-generating tools do not support the in-query options or require all queries to be vendor-independent

Full vs. Partial Plans

Abstract plans can be full plans, describing all query processing steps and options, or they can be partial plans. A partial plan might specify that an index is to be used for the scan of a particular table, without specifying the index name or the join order for the query. For example:

```
select t1.c11, t2.c21
from t1, t2, t3
where t1.c11 = t2.c21
and t1.c11 = t3.c31
```

The full abstract plan includes:

- The join type, either **nl_g_join** for nested-loop joins, or **m_g_join** for merge joins. The plan for this query specifies a nested-loop join.
- The join order, included in the **nl_g_join** clause.
- The type of scan, **t_scan** for table scan or **i_scan** for index scan.
- The name of the index chosen for the tables that are accessed via an index scan.
- The scan properties: the parallel degree, I/O size, and cache strategy for each table in the query.

The abstract plan for the query above specifies the join order, the access method for each table in the query, and the scan properties for each table:

```
( nl_g_join
    ( t_scan t2 )
    ( i_scan t1_c11_ix t1 )
    ( i_scan t3_c31_ix t3 )
)
( prop t3
    ( parallel 1 )
    ( prefetch 16 )
    ( lru )
)
( prop t1
    ( parallel 1 )
    ( prefetch 16 )
    ( lru )
)
( prop t2
    ( parallel 1 )
    ( prefetch 16 )
    ( lru )
)
```

Chapter 24, "Abstract Plan Language Reference," provides a reference to the abstract plan language and syntax.

Creating a Partial Plan

When abstract plans are captured, full abstract plans are generated and stored. You can write partial plans to affect only a subset of the optimizer choices. If the query above had not used the index on *t3*, but all other parts of the query plan were optimal, you could create a partial plan for the query using the **create plan** command. This partial plan specifies only the index choice for *t3*:

```
create plan
"select t1.c11, t2.c21
from t1, t2, t3
where t1.c11 = t2.c21
and t1.c11 = t3.c31"
"( i_scan t3_c31_ix t3 )"
```

You can also create abstract plans with the **plan** clause for **select**, **delete**, **update**, and other optimizable commands. See "Creating Plans Using SQL" on page 22-8.

Abstract Plan Groups

When you first install Adaptive Server, there are two abstract plan groups:

- *ap_stdout*, used by default for capturing plans
- *ap_stdin*, used by default for plan association

A System Administrator can enable server-wide plan capture to *ap_stdout*, so that all query plans for all queries are captured. Server-wide plan association uses queries and plans from *ap_stdin*. If some queries require specially-tuned plans, they can be made available server-wide.

A System Administrator or Database Owner can create additional plan groups, copy plans from one group to another, and compare plans in two different groups.

The capture of abstract plans and the association of abstract plans with queries always happens within the context of the currently-active plan group. Users can use session-level **set** commands to enable plan capture and association.

Some of the ways abstract plan groups can be used are:

- A query tuner can create abstract plans in a group created for testing purposes without affecting plans for other users on the system
- Using plan groups, "before" and "after" sets of plans can be used to determine the effects of system or upgrade changes on query optimization.

See Chapter 22, "Creating and Using Abstract Plans," for information on enabling the capture and association of plans.

How Abstract Plans Are Associated with Queries

When an abstract plan is saved, all white space (returns, tabs, and multiple spaces) in the query is trimmed to a single space, and a hash-key value is computed for the white-space trimmed SQL statement. The trimmed SQL statement and the hash key are stored in *sysqueryplans* along with the abstract plan, a unique plan ID, the user's ID, and the ID of the current abstract plan group.

When abstract plan association is enabled, the hash key for incoming SQL statements is computed, and this value is used to search for the matching query and abstract plan in the current association group, with the corresponding user ID. The full **association key** of an abstract plans consists of:

- The user ID of the current user
- The group ID of the current association group
- The full query text

Once a matching hash key is found, the full text of the saved query is compared to the query to be executed, and used if it matches.

Note that the association key combination of user ID, group ID and query text means that for a given user, there cannot be two queries in the same abstract plan group that have the same query text, but different query plans.

Managing Abstract Plans

A full set of system procedures allows System Administrators and Database Owners to administer plans and plan groups. Individual users can view, drop, and copy the plans for the queries that they have run. See Chapter 23, “Managing Abstract Plans with System Procedures.”

22 Creating and Using Abstract Plans

This chapter provides an overview of the commands used to capture abstract plans and to associate incoming SQL queries with saved plans. Any user can issue session-level commands to capture and load plans during a session, and a System Administrator can enable server-wide abstract plan capture and association. This chapter also describes how to specify abstract plans using SQL.

This chapter contains the following sections:

Using *set* Commands to Capture and Associate Plans

At the session level, any user can enable and disable capture and use of abstract plans with the **set plan dump** and **set plan load** commands. The **set plan replace** command determines whether existing plans are overwritten by changed plans.

Enabling and disabling abstract plan modes takes effect at the end of the batch in which the command is included (similar to **showplan**). Therefore, change the mode in a separate batch before you run your queries:

```
set plan dump on
go
/*queries to run*/
go
```

➤ ***Note***

Any **set plan** commands used in a stored procedure do not affect the procedure in which they are included, but remain in effect after the procedure completes.

Enabling Plan Capture Mode with *set plan dump*

The **set plan dump** command activates and deactivates the capture of abstract plans. You can save the plans to the default group, *ap_stdout*, by using **set plan dump** with no group name:

```
set plan dump on
```

To start capturing plans in a specific abstract plan group, specify the group name. This example sets the group *dev_plans* as the capture group:

```
set plan dump dev_plans on
```

The group that you specify must exist before you issue the **set** command. The system procedure **sp_add_qpgroup** creates abstract plan groups; only the System Administrator or Database Owner can create an abstract plan group. Once an abstract plan group exists, any user can dump plans to the group. See “Creating an Abstract Plan Group” on page 23-2 for information on creating a plan group.

To deactivate the capturing of plans, use:

```
set plan dump off
```

You do not need to specify a group name to end capture mode. Only one abstract plan group can be active for saving or matching abstract plans at any one time. If you are currently saving plans to a group, you must turn off the plan **dump mode**, and reenable it for the new group, as shown here:

```
set plan dump on /*save to the default group*/
go
/*some queries to be captured */
go
set plan dump off
go
set plan dump dev_plans on
go
/*additional queries*/
go
```

The use of the **use database** command while **set plan dump** is in effect disables plan dump mode.

Associating Queries with Stored Plans

The **set plan load** command activates and deactivates the association of queries with stored abstract plans.

To start the association mode using the default group, *ap_stdin*, use the command:

```
set plan load on
```

To enable association mode using another abstract plan group, specify the group name:

```
set plan load test_plans on
```

Only one abstract plan group can be active for plan association at one time. If plan association is active for a group, you must deactivate the current group and start association for the new group, as shown here:

```
set plan load test_plans on
go
/*some queries*/
go
set plan load off
go
set plan load dev_plans on
go
```

The use of the **use database** command while **set plan load** is in effect disables plan load mode.

Using Replace Mode During Plan Capture

While plan capture mode is active, you can choose whether to have plans for the same query replace existing plans by enabling or disabling **set plan replace**. This command activates plan replacement mode:

```
set plan replace on
```

You do not specify a group name with **set plan replace**; it affects the current active capture group.

To disable plan replacement:

```
set plan replace off
```

The use of the **use database** command while **set plan replace** is in effect disables plan replace mode.

When To Use Replace Mode

When you are capturing plans, and a query has the same query text as an already-saved plan, the existing plan is not replaced unless **replace** mode is enabled. If you have captured abstract plans for specific queries, and you are making physical changes to the database that affect optimizer choices, you need to replace existing plans for these changes to be saved.

Some actions that might require plan replacement are:

- Adding or dropping indexes, or changing the keys or key ordering in indexes
- Changing the partitioning on a table
- Adding or removing buffer pools
- Changing configuration parameters that affect query plans

For plans to be replaced, **plan load** mode should not be enabled in most cases. When plan association is active, any plan specifications are used as inputs to the optimizer. For example, if a full query plan includes the **prefetch** property and an I/O size of 2K, and you have created a 16K pool and want to replace the prefetch specification in the plan, do not enable **plan load** mode.

You may want to check query plans and replace some abstract plans as data distribution changes in tables, or after rebuilds on indexes, updating statistics, or changing the locking scheme.

Using Dump, Load, and Replace Modes Simultaneously

You can have both **plan dump** and **plan load** mode active simultaneously, with or without **replace** mode active.

Using *dump* and *load* to the Same Group

If you have enabled **dump** and **load** to the same group, without **replace** mode enabled:

- If a valid plan exists for the query, it is loaded and used to optimize the query.
- If a plan exists that is not valid (say, because an index has been dropped) a new plan is generated and used to optimize the query, but is not saved.
- If the plan is a partial plan, a full plan is generated, but the existing partial plan is not replaced
- If a plan does not exist for the query, a plan is generated and saved.

With **replace** mode also enabled:

- If a valid plan exists for the query, it is loaded and used to optimize the query.
- If the plan is not valid, a new plan is generated and used to optimize the query, and the old plan is replaced.
- If the plan is a partial plan, a complete plan is generated and used, and the existing partial plan is replaced. The specifications in the partial plan are used as input to the optimizer.
- If a plan does not exist for the query, a plan is generated and saved.

Using *dump* and *load* to Different Groups

If you have **dump** enabled to one group, and **load** enabled from another group, without **replace** mode enabled:

- If a valid plan exists for the query in the load group, it is loaded and used. The plan is saved in the dump group, unless a plan for the query already exists in the dump group.
- If the plan in the load group is not valid, a new plan is generated. The new plan is saved in the dump group, unless a plan for the query already exists in the dump group.
- If the plan in the load group is a partial plan, a full plan is generated and saved in the dump group, unless a plan already exists. The specifications in the partial plan are used as input to the optimizer.
- If there is no plan for the query in the load group, the plan is generated and saved in the dump group, unless a plan for the query exists in the dump group.

With **replace** mode active:

- If a valid plan exists for the query in the load group, it is loaded and used.
- If the plan in the load group is not valid, a new plan is generated and used to optimize the query. The new plan is saved in the dump group.
- If the plan in the load group is a partial plan, a full plan is generated and saved in the dump group. The specifications in the partial plan are used as input to the optimizer.
- If a plan does not exist for the query in the load group, a new plan is generated. The new plan is saved in the dump group.

set plan exists check Option

The **exists check** mode can be used during query plan association to speed performance when users require abstract plans for fewer than 20 queries from an abstract plan group. If a small number of queries require plans to improve their optimization, enabling **exists check** mode speeds execution of all queries that do not have abstract plans, because they do not check for plans in *sysqueryplans*.

When **set plan load** and **set exists check** are both enabled, the hash keys for up to 20 queries in the load group are cached for the user. If the load group contains more than 20 queries, **exists check** mode is disabled. Each incoming query is hashed; if its hash key is not stored in the abstract plan cache, then there is no plan for the query. This speeds the execution of all queries that do not have saved plans.

The syntax is:

```
set plan exists check {on | off}
```

You must enable load mode before you enable plan hash-key caching.

A System Administrator can configure server-wide plan hash-key caching with the configuration parameter **abstract plan cache**. To enable server-wide plan caching, use:

```
sp_configure "abstract plan cache", 1
```

Using Other *set* Options with Abstract Plans

You can combine other **set** tuning options c with **set plan dump** and **set plan load**.

Using *showplan*

When **showplan** is turned on, and abstract plan association mode has been enabled with **set plan load**, **showplan** prints the plan ID of the matching abstract plan at the beginning of the **showplan** output for the statement:

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
Optimized using an Abstract Plan (ID : 832005995).
```

If you run queries using the **plan** clause added to a SQL statement, **showplan** displays:

```
Optimized using the Abstract Plan in the PLAN
clause.
```

Using *noexec*

You can use **noexec** mode to capture abstract plans without actually executing the queries. If **noexec** mode is in effect, queries are optimized and abstract plans are saved, but no query results are returned.

To use **noexec** mode while capturing abstract plans, execute any needed procedures (such as **sp_add_qpgroup**) and other **set** options (such as **set plan dump**) before enabling **noexec** mode. The following example shows a typical set of steps:

```
sp_add_qpgroup pubs_dev
go
set plan dump pubs_dev on
go
set noexec on
go
select type, sum(price) from titles group by type
go
```

Using *forceplan*

If **set forceplan on** is in effect, and query association is also enabled for the session, **forceplan** is ignored if a full abstract plan is used to optimize the query. If a partial plan does not completely specify the join order:

- First, the tables in the abstract plan are ordered, as specified.
- The remaining tables are ordered as specified in the **from** clause.
- The two lists of tables are merged.

Server-wide Abstract Plan Capture and Association Modes

A System Administrator can enable server-wide plan capture, association, and replacement modes with these configuration parameters:

- **abstract plan dump** – Enables dumping to the default abstract plans capture group, *ap_stdout*.

- **abstract plan load** – Enables loading from the default abstract plans loading group, *ap_stdin*.
- **abstract plan replace** – When plan dump mode is also enabled, enables plan replacement.
- **abstract plan cache** – Enables caching of abstract plan hash IDs; **abstract plan load** must also be enabled. See “set plan exists check Option” on page 22-6 for more information.

By default, these configuration parameters are set to 0, which means that capture and association modes are off. To enable a mode, set the configuration value to 1:

```
sp_configure "abstract plan dump", 1
```

Enabling any of the server-wide abstract plan modes is dynamic; you do not have to reboot the server.

Server-wide capture and association allows the System Administrator to capture all plans for all users on a server. You cannot override he server-wide modes at the session level.

Creating Plans Using SQL

You can directly specify the abstract plan for a query by:

- Using the **create plan** command
- Adding the plan clause to **select**, **insert...select**, **update**, **delete** and **return** commands, and to **if** and **while** clauses

For information on writing plans, see Chapter 25, “A User’s Guide to Abstract Query Plans.”

Using *create plan*

The **create plan** command specifies the text of a query, and the abstract plan to save for the query.

This example creates an abstract plan:

```
create plan
    "select avg(price) from titles"
"( plan
    ( i_scan type_price_ix titles )
    ( )
)"
```

The plan is saved in the current active plan group. You can also specify the group name:

```
create plan
    "select avg(price) from titles"
"( plan
    ( i_scan type_price_ix titles )
    ( )
)"
into dev_plans
```

If a plan already exists for the specified query in the current plan group, or the plan group that you specify, you must first enable **replace** mode in order to overwrite the existing plan.

If you want to see the plan ID that is used for a plan you create, **create plan** can return the ID as a variable. You must declare the variable first. This example returns the plan ID:

```
declare @id int
create plan
    "select avg(price) from titles"
"( plan
    ( i_scan type_price_ix titles )
    ( )
)"
into dev_plans
and set @id
select @id
```

When you use **create plan**, the query in the plan is not executed. This means that:

- The text of the query is not parsed, so the query is not checked for valid SQL syntax.
- The plans are not checked for valid abstract plan syntax.
- The plans are not checked to determine whether they are compatible with the SQL text.

To guard against errors and problems, you should immediately execute the specified query with **showplan** enabled.

Using the *plan* Clause

You can use the **plan** clause with the following SQL statements to specify the plan to use to execute the query:

- **select**

- **insert...select**
- **delete**
- **update**
- **if**
- **while**
- **return**

This example specifies the plan to use to execute the query:

```
select avg(price) from titles
     plan
" ( plan
    ( i_scan type_price_ix titles )
    ( )
)"
```

When you specify an abstract plan for a query, the query is executed using the specified plan. If you have **showplan** enabled, this message is printed:

```
Optimized using the Abstract Plan in the PLAN
clause.
```

When you use the **plan** clause with a query, any errors in the SQL text, the plan syntax, and any mismatches between the plan and the SQL text are reported as errors. For example, this plan omits the empty parentheses that specify the step of returning the aggregate:

```
/* step missing! */
select avg(price) from titles
     plan
" ( plan
    ( i_scan type_price titles )
)"
```

It returns the following message:

```
Msg 1005, Level 16, State 1:
Server ‘smj’, Line 2:
Abstract Plan (AP) : The number of operands of the
PLAN operator in the AP differs from the number of
steps needed to compute the query. The extra items
will be ignored. Check the AP syntax and its
correspondence to the query.
```

Plans specified with the **plan** clause are saved in *sysqueryplans* only if plan capture is enabled. If a plan for the query already exists in the current capture group, you must enable **replace** mode in order to replace an existing plan.

23 Managing Abstract Plans with System Procedures

System Procedures for Managing Abstract Plans

The system procedures for managing abstract plans work on individual plans or on abstract plan groups. This chapter provides an introduction to the basic functionality and use of the system procedures for working with abstract plans. For detailed information on each procedure, see the *Adaptive Server Reference Manual*.

Managing an Abstract Plan Group

You can use system procedures to create, drop, rename, and provide information about an abstract plan group.

Creating an Abstract Plan Group

sp_add_qpgroup creates and names an abstract plan group. Unless you are using the default capture group, *ap_stdout*, you must create a plan group before you can begin capturing plans. For example, to start saving plans in a group called *dev_plans*, you must create the group, then issue the **set plan dump** command, specifying the group name:

```
sp_add_qpgroup dev_plans
set plan dump dev_plans on
/*SQL queries to capture*/
```

Only a System Administrator or Database Owner can add abstract plan groups. Once a group is created, any user can dump or load plans from the group.

Dropping an Abstract Plan Group

sp_drop_qpgroup drops an abstract plan group.

The following restrictions apply to **sp_drop_qpgroup**:

- Only a System Administrator or Database Owner can drop abstract plan groups.
- You cannot drop a group that contains plans. To remove all plans from a group, use **sp_drop_all_qplans**, specifying the group name.
- You cannot drop the default abstract plan groups *ap_stdin* and *ap_stdout*.

This command drops the *dev_plans* plan group:

```
sp_drop_qpgroup dev_plans
```

Getting Information about an Abstract Plan Group

sp_help_qpgroup prints information about an abstract plan group, or about all abstract plan groups in a database.

When you use **sp_help_qpgroup** without a group name, it prints the names of all abstract plan groups, the group IDs, and the number of plans in each group:

```
sp_help_qpgroup
```

```
Query plan groups in database ‘pubtune’
 Group                          GID         Plans
 ------------------------------ ----------- -----------
 ap_stdin                                 1           0
 ap_stdout                                2           2
 p_prod                                   4           0
 priv_test                                8           1
 ptest                                    3          51
 ptest2                                   7         189
```

When you use **sp_help_qpgroup** with a group name, the report provides statistics about plans in the specified group. This example reports on the group *ptest2*:

```
sp_help_qpgroup ptest2
```

```
Query plans group 'ptest2', GID 7

 Total Rows  Total QueryPlans
 ----------- ----------------
         452              189
sysqueryplans rows consumption, number of query
plans per row count
 Rows        Plans
 ----------- -----------
           5           2
           3          68
           2         119
Query plans that use the most sysqueryplans rows
 Rows        Plan
 ----------- -----------
           5  1932533918
           5  1964534032
 Hashkeys
 -----------
         123
There is no hash key collision in this group.
```

When reporting on an individual group, **sp_help_qpgroup** reports:

- The total number of abstract plans, and the total number of rows in the *sysqueryplans* table.
- The number of plans that have multiple rows in *sysqueryplans*. They are listed in descending order, starting with the plans with the largest number of rows.
- Information about the number of hash keys and hash-key collisions. Abstract plans are associated with queries by a hashing algorithm over the entire query.

When a System Administrator or the Database Owner executes **sp_help_qpgroup**, the procedure reports on all of the plans in the database or in the specified group. When any other user executes **sp_help_qpgroup**, it reports only on plans that he or she owns.

sp_help_qpgroup provides several report modes. The report modes are:

Mode	Information Returned
full	The number of rows and number of plans in the group, the number of plans that use two or more rows, the number of rows and plan IDs for the longest plans, and number of hash keys, and has- key collision information. This is the default report mode.
stats	All of the information from the full report, except hash-key information.
hash	The number of rows and number of abstract plans in the group, the number of hash keys, and hash-key collision information.
list	The number of rows and number of abstract plans in the group, and the following information for each query / plan pair: hash key, plan ID, first few characters of the query, and the first few characters of the plan.
queries	The number of rows and number of abstract plans in the group, and the following information for each query: hash key, plan ID, first few characters of the query.
plans	The number of rows and number of abstract plans in the group, and the following information for each plan: hash key, plan ID, first few characters of the plan.
counts	The number of rows and number of abstract plans in the group, and the following information for each plan: number of rows, number of characters, hash key, plan ID, first few characters of the query.

This example shows the output for the **counts** mode:

```
sp_help_qpgroup ptest1, counts
```

```
Query plans group 'ptest1', GID 3

 Total Rows  Total QueryPlans
 ----------- ----------------
          48               19

Query plans in this group

Rows  Chars     hashkey     id          query
----- --------- ----------- ----------- ----------------------------
```

```
3      623  1801454852   876530156 select title from titles ...
3      576   476063777   700529529 select au_lname, au_fname...
3      513   444226348   652529358 select au1.au_lname, au1....
3      470   792078608   716529586 select au_lname, au_fname...
3      430   789259291   684529472 select au1.au_lname, au1....
3      425  1929666826   668529415 select au_lname, au_fname...
3      421   169283426   860530099 select title from titles ...
3      382   571605257   524528902 select pub_name from publ...
3      355   845230887   764529757 delete salesdetail where ...
3      347   846937663   796529871 delete salesdetail where ...
2      379  1400470361   732529643 update titles set price =...
```

Renaming an Abstract Plan Group

A System Administrator or Database Owner can rename an abstract plan group with **sp_rename_qpgroup**. This example changes the name of the group from *dev_plans* to *prod_plans*:

```
sp_rename_qpgroup dev_plans, prod_plans
```

The new group name cannot be the name of an existing group.

Finding Abstract Plans

sp_find_qplan searches both the query text and the plan text to find plans that match a given pattern.

This example finds all plans where the query includes the string "from titles":

```
sp_find_qplan "%from titles%"
```

This example searches for all abstract plans that perform a table scan:

```
sp_find_qplan "%t_scan%"
```

When a System Administrator or Database Owner executes **sp_find_qplan**, the procedure examines and reports on plans owned by all users. When other users execute the procedure, it searches and reports on only plans that they own.

If you want to search just one abstract plan group, specify the group name with **sp_find_qplan**. This example searches only the *test_plans* group, finding all plans that use a particular index:

```
sp_find_qplan "%i_scan title_id_ix%", test_plans
```

For each matching plan, **sp_find_qplan** prints the group ID, plan ID, query text, and abstract plan text.

Managing Individual Abstract Plans

You can use system procedures to print the query and text of individual plans, to copy, drop, or compare individual plans, or to change the plan associated with a particular query.

Viewing a Plan

sp_help_qplan reports on individual abstract plans. It provides three types of reports that you can specify: **brief**, **full**, and **list**. The **brief** report prints only the first 78 characters of the query and plan; use **full** to see the entire query and plan, or **list** to display only the first 20 characters of the query and plan.

This example prints the default brief report:

```
sp_help_qplan 588529130
```

```
gid         hashkey     id
----------- ----------- -----------
          8  1460604254   588529130
query
-----------------------------------------------------------------
select min(price) from titles
plan
-----------------------------------------------------------------
( plan
     ( i_scan type_price titles )
     ( )
)
( prop titles
     ( parallel ...
```

A System Administrator or Database Owner can use **sp_help_qplan** to report on any plan in the database. Other users can only view the plans that they own.

sp_help_qpgroup reports on all plans in a group. For more information see "Getting Information about an Abstract Plan Group" on page 23-2.

Copying a Plan to Another Group

sp_copy_qplan copies an abstract plan from one group to another existing group. This example copies the plan with plan ID 316528161 from its current group to the *prod_plans* group:

```
sp_copy_qplan 316528161, prod_plans
```

sp_copy_qplan checks to make sure that the query does not already exist in the destination group. If a possible conflict exists, it runs **sp_cmp_qplans** to check plans in the destination group. In addition to the message printed by **sp_cmp_qplans**, **sp_copy_qplan** prints messages when:

- The query and plan you are trying to copy already exists in the destination group
- Another plan in the group has the same user ID and hash key
- Another plan in the group has the same hash key, but the queries are different

If there is a hash-key collision, the plan is copied. If the plan already exists in the destination group or if it would give an association key collision, the plan is not copied. The messages printed by **sp_copy_qplan** contain the plan ID of the plan in the destination group, so you can use **sp_help_qplan** to check the query and plan.

A System Administrator or the Database Owner can copy any abstract plan. Other users can copy only plans that they own. The original plan and group are not affected by **sp_copy_qplan**. The copied plan is assigned a new plan ID, the ID of the destination group, and the user ID of the user who ran the query that generated the plan.

Dropping an Individual Abstract Plan

sp_drop_qplan drops individual abstract plans. This example drops the specified plan:

```
sp_drop_qplan 588529130
```

A System Administrator or Database Owner can drop any abstract plan in the database. Other users can drop only plans that they own.

To find abstract plan IDs, use **sp_find_qplan** to search for plans using a pattern from the query or plan, or **sp_help_qpgroup** to list the plans in a group.

Comparing Two Abstract Plans

Given two plan IDs, **sp_cmp_qplans** compares two abstract plans and the associated queries. For example:

```
sp_cmp_qplans 588529130, 1932533918
```

sp_cmp_qplans prints one message reporting the comparison of the query, and a second message about the plan, as follows:

- For the two queries, one of:
  - The queries are the same.
  - The queries are different.
  - The queries are different but have the same hash key.
- For the plans:
  - The query plans are the same.
  - The query plans are different.

This example compares two plans where the queries and plans both match:

```
sp_cmp_qplans 411252620, 1383780087
```

```
The queries are the same.
The query plans are the same.
```

This example compares two plans where the queries match, but the plans are different:

```
sp_cmp_qplans 2091258605, 647777465
```

```
The queries are the same.
The query plans are different.
```

sp_cmp_qplans returns a status value showing the results of the comparison. The status values are shown in Table 23-1.

Table 23-1: Return status values for sp_cmp_qplans

Return Value	Meaning
0	The query text and abstract plans are the same.
+1	The queries and hash keys are different.
+2	The queries are different, but the hash keys are the same.
+10	The abstract plans are different.
100	One or both of the plan IDs does not exist.

A System Administrator or Database Owner can compare any two abstract plans in the database. Other users can compare only plans that they own.

Changing an Existing Plan

sp_set_qplan changes the abstract plan for an existing plan ID without changing the ID or the query text. It can be used only when the plan text is 255 or fewer characters.

```
sp_set_qplan 588529130, "( i_scan title_ix titles)"
```

A System Administrator or Database Owner can change the abstract plan for any saved query. Other users can modify only plans that they own.

When you execute **sp_set_qplan**, the abstract plan is not checked against the query text to determine whether the new plan is valid for the query, or whether the tables and indexes exist. To test the validity of the plan, execute the associated query.

You can also use **create plan** and the **plan** clause to specify the abstract plan for a query. See “Creating Plans Using SQL” on page 22-8.

Managing All Plans in a Group

These system procedures help manage groups of plans:

- **sp_copy_all_qplans**
- **sp_cmp_all_qplans**
- **sp_drop_all_qplans**

Copying All Plans in a Group

sp_copy_all_qplans copies all of the plans in one abstract plan group to another group. This example copies all of the plans from the *test_plans* group to the *helpful_plans* group:

```
sp_copy_all_qplans test_plans, helpful_plans
```

The *helpful_plans* group must exist before you execute **sp_copy_all_qplans**. It can contain other plans.

sp_copy_all_qplans copies each plan in the group by executing **sp_copy_qplan**, so copying a plan may fail for the same reasons that **sp_copy_qplan** might fail. See “Comparing Two Abstract Plans” on page 23-7.

Each plan is copied as a separate transaction, and failure to copy any single plan does not cause **sp_copy_all_qplans** to fail. If **sp_copy_all_qplans** fails for any reason, and has to be restarted, you see a set of messages for the plans that have already been successfully copied, telling you that they exist in the destination group.

A new plan ID is assigned to each copied plan. The copied plans have the original user's ID. To copy abstract plans and assign new user IDs, you must use **sp_export_qpgroup** and **sp_import_qpgroup**. See "Importing and Exporting Groups of Plans" on page 23-13.

A System Administrator or Database Owner can copy all plans in the database. Other users can copy only plans that they own.

Comparing All Plans in a Group

sp_cmp_all_qplans compares all abstract plans in two groups and reports:

- The number of plans that are the same in both groups
- The number of plans that have the same association key, but different abstract plans
- The number of plans that are present in one group, but not the other

This example compares the plans in *ap_stdout* and *ap_stdin*:

```
sp_cmp_all_qplans ap_stdout, ap_stdin
```

```
If the two query plans groups are large, this
might take some time.
Query plans that are the same
 count
 -----------
         338
Different query plans that have the same
association key

 count
 -----------
          25
Query plans present only in group 'ap_stdout' :

 count
 -----------
           0
Query plans present only in group 'ap_stdin' :

 count
 -----------
           1
```

With the additional specification of a report-mode parameter, **sp_cmp_all_qplans** provides detailed information, including the IDs, queries, and abstract plans of the queries in the groups. The mode parameter lets you get the detailed information for all plans, or just those with specific types of differences.Table 23-2 shows the report modes and what type of information is reported for each mode.

Table 23-2: Report modes for sp_cmp_all_qplans

Mode	Reported Information
counts	The counts of: plans that are the same, plans that have the same association key, but different groups, and plans that exist in one group, but not the other. This is the default report mode.
brief	The information provided by counts, plus the IDs of the abstract plans in each group where the plans are different, but the association key is the same, and the IDs of plans that are in one group, but not in the other.
same	All counts, plus the IDs, queries, and plans for all abstract plans where the queries and plans match.
diff	All counts, plus the IDs, queries, and plans for all abstract plans where the queries and plans are different.
first	All counts, plus the IDs, queries, and plans for all abstract plans that are in the first plan group, but not in the second plan group.
second	All counts, plus the IDs, queries, and plans for all abstract plans that are in the second plan group, but not in the first plan group.
offending	All counts, plus the IDs, queries, and plans for all abstract plans that have different association keys or that do not exist in both groups. This is the combination of the diff, first, and second modes.
full	All counts, plus the IDs, queries, and plans for all abstract plans. This is the combination of same and offending modes.

This example shows the **brief** report mode:

```
sp_cmp_all_qplans ptest1, ptest2, brief
```

```
If the two query plans groups are large, this
might take some time.
Query plans that are the same
 count
 -----------
          39
Different query plans that have the same
association key

 count
 -----------
           4

    ptest1    ptest2

 id1         id2
 ----------- -----------
   764529757  1580532664
   780529814  1596532721
   796529871  1612532778
   908530270  1724533177
Query plans present only in group 'ptest1' :

 count
 -----------
           3


 id
 -----------
   524528902
  1292531638
  1308531695

Query plans present only in group 'ptest2' :

 count
 -----------
           1


 id
 -----------
  2108534545
```

Dropping All Abstract Plans in a Group

sp_drop_all_qplans drops all abstract plans in a group. This example drops all abstract plans in the *dev_plans* group:

```
sp_drop_all_qplans dev_plans
```

When a System Administrator or the Database Owner executes **sp_drop_all_qplans**, all plans belonging to all users are dropped from the specified group. When another user executes this procedure, it affects only the plans owned by that users.

Importing and Exporting Groups of Plans

sp_export_qpgroup and **sp_import_qpgroup** copy groups of plans between *sysqueryplans* and a user table. This allows a System Administrator or Database Owner to:

- Copy abstract plans from one database to another on the same server
- Create a table that can be copied out of the current server with **bcp**, and copied into another server
- Assign different user IDs to existing plans in the same database

Exporting Plans to a User Table

sp_export_qpgroup copies all plans for a specific user from an abstract plan group to a user table. This example copies plans owned by the Database Owner (dbo) from the *fast_plans* group, creating a table called *transfer*:

```
sp_export_qpgroup dbo, fast_plans, transfer
```

sp_export_qpgroup uses **select...into** to create a table with the same columns and datatypes as *sysqueryplans*. If you do not have the **select into/bulkcopy/pllsort** option enabled in the database, you can specify the name of another database. This command creates the export table in *tempdb*:

```
sp_export_qpgroup mary, ap_stdout, "tempdb..mplans"
```

The table can be copied out using **bcp**, and copied into a table on another server. The plans can also be imported to *sysqueryplans* in another database on the same server, or the plans can be imported into *sysqueryplans* in the same database, with a different group name or user ID.

Importing Plans from a User Table

sp_import_qpgroup copies plans from tables created by **sp_export_qpgroup** into a group in *sysqueryplans*. This example copies the plans from the table *tempdb..mplans* into *ap_stdin*, assigning the user ID for the Database Owner:

```
sp_import_qpgroup "tempdb..mplans", dbo, ap_stdin
```

You cannot copy plans into a group that already contains plans for the specified user.

24 Abstract Plan Language Reference

This chapter describes the operators and other language elements in the abstract plan language.

Keywords

The following words are keywords in the abstract query plan language. They are not reserved words, and do not conflict with the names of tables or indexes used in abstract plans. For example, a table or index may be named *hints*.

Operands

The following operands are used in the abstract plan syntax statements:

Table 24-1: Identifiers used

Identifier	Describes
table_name	The name of a base table, that is, a user or system table
correlation_name	The correlation name specified for a table in a query
derived_table	A table that results from the scan of a stored table
stored_table	A base table or a worktable
worktable_name	The name of a worktable
view_name	The name of a view
index_name	The name of an index
subquery_id	An integer identifying the order of the subqueries in the query

table_name and *view_name* can be specified using the notation *database.owner.object_name*.

Derived Tables

A derived table is a result of access to a stored table during query execution. It can be:

- The result set generated by the query
- An intermediate result during query execution; that is, the result of the join of the first two tables in the join order, which is then joined with a third table

Derived tables result from one of the scan operators that specify the access method: **scan**, **i_scan**, or **t_scan**, for example, **(i_scan title_id_ix titles)**.

Schema for Examples

To simplify the sample abstract plan examples, the following tables are used in this section:

```
create table t1 (c11 int, c12 int)
create table t2 (c21 int, c22 int)
```

```
create table t3 (c31 int, c32 int)
```

The following indexes are used:

```
create index i_c11 on t1(c11)
create index i_c12 on t1(c12)
create index i_c11_c12 on t1(c11, c12)
create index i_c21 on t2(c21)
create index i_c22 on t2(c22)
create index i_c31 on t3(c31)
create index i_c32 on t3(c32)
```

g_join

Function

Specifies the join of two or more derived tables without specifying the join type (nested-loop or sort-merge).

Syntax

```
( g_join derived_table1 derived_table2
   )

( g_join   ( derived_table1 )
           ( derived_table2 )
           ...
           ( derived_tableN )
   )
```

Operands

derived_table1...derived_tableN – are the derived tables to be united.

Result

A derived table that is the join of the specified derived tables.

Side Effects

The tables are joined in the order specified in the **g_join** clause.

Examples

1.
```
select *
from t1, t2
where c21 = 0
and c22 = c12
```
```
( g_join
    ( i_scan i_c21 t2 )
    ( i_scan i_c12 t1 )
)
```
   Table *t2* is the outer table, and *t1* the inner table in the join order.

2.
```
select *
from t1, t2, t3
where c21 = 0
and c22 = c12
and c11 = c31
```

```
( g_join
    ( i_scan i_c21 t2 )
    ( i_scan i_c12 t1 )
    ( i_scan i_c31 t3 )
)
```

Table *t2* is joined with *t1*, and the derived table is joined with *t3*.

Comments

- The **g_join** operator is a generic logical operator that describes all binary joins (inner join, outer join, or existence join).
- The **g_join** operator is never used in generated plans; **nl_g_join** and **m_g_join** operators indicate the join type used.
- The optimizer chooses between a nested-loop join and a sort-merge join when the **g_join** operator is used. To specify a sort-merge join, use **m_g_join**. To specify a nested-loop join, use **nl_g_join**.
- The syntax provides a shorthand method of described a join involving multiple tables. This syntax:

```
( g_join
        ( scan t1)
        ( scan t2)
        ( scan t3)
        ...
        ( scan tN-1)
        ( scan tN)
)
```

  is shorthand for:

```
( g_join
        ( g_join
                ...
                    ( g_join
                            (g_join
                                    ( scan t1)
                                    ( scan t2)
                            )
                            ( scan t3)
                    )
                ...
                ( scan tN-1)
        )
        ( scan tN)
)
```

- If **g_join** is used to specify the join order for some, but not all, of the tables in a query, the optimizer uses the join order specified, but may insert other tables between the **g_join** operands. For example, for this query:

```
select *
    from t1, t2, t3
    where ...
```

the following partial abstract plan describes only the join order of *t1* and *t2*:

```
( g_join
        ( scan t2)
        ( scan t1)
)
```

The optimizer can choose any of the three join orders: *t3*-*t2*-*t1*, *t2*-*t3*-*t1* or *t2*-*t1*-*t3*.

- If **set forceplan on** is in effect, and query association is also enabled for the session, **forceplan** is ignored if a full abstract plan is used to optimize the query. If a partial plan does not completely specify the join order:
  - First, the tables in the abstract plan are ordered as specified.
  - The remaining tables are ordered as specified in the **from** clause.
  - The two lists of tables are merged.

See Also

m_g_join, **nl_g_join**

hints

Function

Introduces and groups items in a partial abstract plan.

Syntax

```
( hints ( derived_table )
        ...
   )
```

Operands

derived_table – is one or more expressions that generate a derived table.

Result

A derived table.

Side Effects

The specified hints are used during query optimization.

Examples

1.
```
select *
from t1, t2
where c12 = c21
    and c11 > 0
    and c22 < 1000
```
```
( hints
    ( g_join
        ( t_scan t2 )
        ( i_scan () t1 )
    )
)
```

Specifies a partial plan, including a table scan on *t2*, the use of some index on *t1*, and the join order *t1*-*t2*. The index choice for *t1* and the type of join (nested-loop or sort-merge) is left to the optimizer.

Comments

- The **hints** operator appears as the root of a partial abstract plan that includes multiple steps. If a partial plan contains only one expression, **hints** is optional.

- The **hints** operator does not appear in plans generated by the optimizer; these are always full plans.
- Hints can be associated with queries:
  - By changing the plan for an existing query with **sp_set_qplan**.
  - By specifying the plan for a query with the **plan** clause. To save the query and hints, **set plan dump** must be enabled.
  - By using the **create plan** command.
- When hints are specified in the **plan** clause for a SQL statement, the plans are checked to be sure they are valid. When hints are specified using **sp_set_qplan**, plans are not checked before being saved.

i_scan

Function

Specifies an index scan of a base table.

Syntax

(i_scan *index_name base_table*)

(i_scan () *base_table*)

Operands

index_name – is the name or index ID of the index to use for an index scan of the specified stored table. Use of empty parentheses specify that an index scan (rather than table scan) is to be performed, but leaves the choice of index to the optimizer.

base_table – is the name of the base table to be scanned.

Result

A derived table produced by a scan of the base table.

Side Effects

The index is used to scan the table, or, if no index is specified, an index is used rather than a table scan.

Examples

1. **select * from t1 where c11 = 0**

   ```
   ( i_scan i_c11 t1 )
   ```

   Specifies the use of index *i_c11* for a scan of *t1*.

2. ```
   select *
   from t1, t2
   where c11 = 0
       and c22 = 1000
       and c12 = c21
   ```

   ```
   ( g_join
           ( scan t2 )
           ( i_scan () t1 )
   )
   ```

   Specifies a partial plan, indicating the join order, but allowing the optimizer to choose the access method for *t2*, and the index for *t1*.
   ```

3. **select * from t1 where c12 = 0**

```
( i_scan 2 t1 )
```

Identifies the index on *t1* by index ID, rather than by name.

Comments

- Use of empty parentheses after the **i_scan** operator allows the optimizer to choose the index or to perform a table scan if no index exists on the table.
- When the **i_scan** operator is specified, a covering index scan is always performed when all of the required columns are included in the index. No abstract plan specification is needed to describe a covering index scan.
- Use of the **i_scan** operator suppresses the choice of the reformatting strategy and the OR strategy, even if the specified index does not exist. The optimizer chooses another useful index and an advisory message is printed. If no index is specified for **i_scan**, or if no indexes exist, a table scan is performed, and an advisory message is printed.
- Although specifying an index using the index ID is valid in abstract query plans, using an index ID is not recommended. If indexes are dropped and re-created in a different order, plans become invalid or perform suboptimally.

See Also

scan, **t_scan**

in

Function

Identifies the location of a table that is specified in a subquery or view.

Syntax

```
( in ( [ subq subquery_id | view view_name ] )
   )
```

Operands

subq *subquery_id* – is an integer identifying a subquery. In abstract plans, subquery numbering is based on the order of the leading open parentheses for the subqueries in a query.

view *view_name* – is the name of a view. The specification of database and owner name in the abstract plan must match the usage in the query in order for plan association to be performed.

Side Effects

Identifies the occurrence of a table in a view or subquery of the SQL query.

Examples

1.
```
create view v1 as
select * from t1

select * from v1
```

```
( t_scan ( table t1 ( in ( view v1 ) ) ) )
```

Identifies the view in which table *t1* is used.

2.
```
select *
from t2
where c21
in (select c12 from t1)
```

```
( g_join
    ( t_scan t2 )
    ( t_scan ( table t1 ( in ( subq 1 ) ) ) )
)
```

Identifies the scan of table *t1* in subquery 1.

3.
```
create view v9
as
select *
from t1
where c11 in (select c21 from t2)

create view v10
as
select * from v9
where c11 in (select c11 from v9)

select * from v10, t3
where c11 in
        (select c11 from v10 where c12 = t3.c31)
```

```
( g_join
( t_scan t3 )
( i_scan i_c21 ( table t2 ( in ( subq 1 ) ( view v9 ) ( view v10 ))))
( i_scan i_c11 ( table t1 ( in ( view v9 ) ( view v10 ))))
( i_scan i_c11 ( table t1 ( in ( view v9 ) ( view v10 ) ( subq 1 ))))
( i_scan i_c11 ( table t1 ( in ( view v9 ) ( subq 1 ) ( view v10 ))))
( i_scan i_c21 ( table t2 ( in ( subq 1 ) ( view v9 ) ( subq 1 ) (
view v10 ))))
( i_scan i_c11 ( table t1 ( in ( view v9 ) ( subq 1 ) ( view v10 ) (
subq 1 ))))
( i_scan i_c21 ( table t2 ( in ( subq 1 ) ( view v9 ) ( view v10 ) (
subq 1 ))))
( i_scan i_c21 ( table t2(in( subq 1 )( view v9 )( subq 1 )( view v10
) ( subq 1))))
)
```

An example of multiple nesting of views and subqueries.

Comments

- The **in** list has the innermost items to the left, near the table's name (itself the deeply nested item), and the outermost items (the ones occurring in the top level query) to the right. For example, the qualification:

```
(table t2 (in (subq 1) (view v9) (subq 1) (view v10) (subq 1) ) )
```

can be read in either direction:

- Reading left to right, starting from the table: the base table *t2* as scanned in the first subquery of view *v9*, which occurs in the first subquery of view *v10*, which occurs in the first subquery of the main query
- Reading from right to left, that is, starting from the main query: in the main query there's a first subquery, that scans the view

v10, that contains a first subquery that scans the view *v9*, that contains a first subquery that scans the base table *t2*

See Also

nested, **subq**, **table**, **view**

lru

Function

Specifies LRU cache strategy for the scan of a stored table.

Syntax

```
( prop table_name
       ( lru )
   )
```

Operands

table_name – is the table to which the property is to be applied.

Side Effects

LRU strategy is used in the resulting query plan.

Examples

1. **select * from t1**

```
( prop t1
        ( lru)
)
```

Specifies the use of LRU cache strategy for the scan of *t1*.

Comments

- Partial plans can specify scan properties without specifying other portions of the query plan.
- Full query plans always include all scan properties.

See Also

mru, **prop**

m_g_join

Function

Specifies a merge join of two derived tables.

Syntax

```
( m_g_join (
   ( derived_table1 )
   ( derived_table2 )
   )
```

Operands

derived_table1...derived_tableN – are the derived tables to be united. *derived_table1* is always the outer table and *derived_table2* is the inner table.

Result

A derived table that is the join of the specified derived tables.

Side Effects

The tables are joined in the order specified in the **m_g_join** clause.

Examples

1.
```
select t1.c11, t2.c21
from t1, t2, t3
where t1.c11 = t2.c21
    and t1.c11 = t3.c31
```
```
( nl_g_join
        ( m_g_join
            ( i_scan i_c31 t3 )
            ( i_scan i_c11 t1 )
        )
        ( t_scan t2 )
)
```
Specifies a right-merge join of tables *t1* and *t3*, followed by a nested-loop join with table *t2*.

2.
```
select * from t1, t2, t3
where t1.c11 = t2.c21 and t1.c11 = t3.c31
and t2.c22 =7
```

```
( nl_g_join
    ( m_g_join
        ( i_scan i_c21 t2 )
        ( i_scan i_c11 t1 )
    )
    ( i_scan i_c31 t3 )
)
```

Specifies a full-merge join of tables *t2* (outer) and *t1* (inner), followed in the join order by a nested-loop join with *t3*.

3. **select c11, c22, c32**
   from t1, t2, t3
   where t1.c11 = t2.c21
   and t2.c22 = t3.c32

```
( m_g_join
        (nl_g_join
            (i_scan i_c11 t1)
            (i_scan i_c12 t2)
        )
    (i_scan i_c32_ix t3)
)
```

Specifies a nested-loop join of *t1* and *t2*, followed by a merge join with *t3*.

Comments

- The sort step and worktable required to process sort-merge join queries are not represented in abstract plans.
- If the **m_g_join** operator is used to specify a join that cannot be performed as a merge join, the specification is silently ignored.

See Also

g_join, **nl_g_join**

mru

Function

Specifies MRU cache strategy for the scan of a stored table.

Syntax

```
( prop table_name
       ( mru)
   )
```

Operands

table_name – is the table to which the property is to be applied.

Side Effects

MRU strategy is specified in the resulting query plan.

Examples

1. **select * from t1**

```
( prop t1
        ( mru )
)
```

Specifies the use of MRU cache strategy for the table.

Comments

- Partial plans can specify scan properties without specifying other portions of the query plan.
- Generated query plans always include all scan properties.
- If **sp_cachestrategy** has been used to disable MRU replacement for a table or index, and the query plan specifies MRU, the specification in the abstract plan is silently ignored.

See Also

lru, **prop**

nested

Function

Describes the nesting of subqueries on a derived table.

Syntax

```
( nested
       ( derived_table )
           ( subquery_specification )
   )
```

Operands

derived_table – is the derived table over which to nest the specified subquery.

subquery_specification – is the subquery to nest over *derived_table*.

Result

A derived table.

Side Effects

The subquery is executed at the specified attachment point in the query plan.

Examples

1. **select c11 from t1**
   where c12 =
   (select c21 from t2 where c22 = t1.c11)

```
( nested
        ( t_scan t1 )
        ( subq 1
                ( t_scan ( table t2 ( in ( subq 1 ) ) ) )
        )
)
```

A single nested subquery.

2. **select c11 from t1**
   where c12 =
   (select c21 from t2 where c22 = t1.c11)
   and c12 =
   (select c31 from t3 where c32 = t1.c11)

```
( nested
        ( nested
                ( t_scan t1 )
                ( subq 1
                        ( t_scan ( table t2 ( in ( subq 1 ) ) ) )
                )
        )
        ( subq 2
                ( t_scan ( table t3 ( in ( subq 2 ) ) ) )
        )
)
```

The two subqueries are both nested in the main query.

3. **select c11 from t1**
   where c12 =
   (select c21 from t2 where c22 =
   (select c31 from t3 where c32 = t1.c11))

```
( nested
        ( t_scan t1 )
        ( subq 1
                ( nested
                        ( t_scan ( table t2 ( in ( subq 1 ) ) ) )
                        ( subq 2
                              ( t_scan ( table t3 ( in ( subq 2 ) ) ) )
                        )
                )
        )
)
```

A level 2 subquery nested into a level 1 subquery nested in the main query.

Comments

- Materialized and flattened subqueries do not appear under a **nested** operator. See **subq** on page 24-34 for examples.

See Also

in, subq

nl_g_join

Function

Specifies a nested-loop join of two or more derived tables.

Syntax

```
( nl_g_join( derived_table1 )
           ( derived_table2 )
           ...
           ( derived_tableN )
   )
```

Operands

derived_table1...*derived_tableN* – are the derived tables to be united.

Result

A derived table that is the join of the specified derived tables.

Side Effects

The tables are joined in the order specified in the **nl_g_join** clause.

Examples

1.
```
select *
from t1, t2
where c21 = 0
and c22 = c12
```

```
( nl_g_join
    ( i_scan i_c21 t2 )
    ( i_scan i_c12 t1 )
)
```

   Table *t2* is the outer table, and *t1* the inner table in the join order.

2.
```
select *
from t1, t2, t3
where c21 = 0
and c22 = c12
and c11 = c31
```

```
( nl_g_join
    ( i_scan i_c21 t2 )
    ( i_scan i_c12 t1 )
    ( i_scan i_c31 t3 )
)
```

Table *t2* is joined with *t1*, and the derived table is joined with *t3*.

Comments

- The **nl_g_join** operator is a generic logical operator that describes all binary joins (inner join, outer join, or semijoin). The joins are performed using the nested-loops query execution method.

See Also

g_join, **m_g_join**

parallel

Function

Specifies the degree of parallelism for the scan of a stored table.

Syntax

```
( prop table_name
       ( parallel degree )
   )
```

Operands

table_name – is the table to which the property is to be applied.

degree – is the degree of parallelism to use for the scan.

Side Effects

The scan is performed using the specified number of worker processes, if available.

Examples

1. **select * from t1**

```
(prop t1
        ( parallel 5 )
)
```

Specifies that 5 worker processes should be used for the scan of the *t1* table.

Comments

- Partial plans can specify scan properties without specifying other portions of the query plan.
- If a saved plan specifies the use of a number of worker processes, but session-level or server-level values are different when the query is executed:
  - If the plan specifies more worker processes than permitted by the current settings, the current settings are used or the query is executed using a serial plan.
  - If the plan specifies fewer worker processes than permitted by the current settings, the values in the plan are used.

These changes to the query plan are performed transparently to the user, so no warning messages are issued.

See Also

prop

plan

Function

Provides a mechanism for grouping the query plan steps of multi-step queries, such as queries requiring worktables, and queries computing aggregate values.

Syntax

```
(plan
       query_step1
       ...
       query_stepN
   )
```

Operands

query_step1...query_stepN – specify the abstract plan steps for the execution of each step in the query.

Result

A derived table.

Side Effects

Tables are accessed in the order specified, with the specified access methods.

Examples

1. **select max(c11) from t1 group by c12**

```
( plan
    ( store Worktab1
        ( t_scan t1 )
    )
    ( t_scan ( work_t Worktab1 ) )
)
```

Returns a vector aggregate. The first operand of the plan operator creates *Worktab1* and specifies a table scan of the base table. The second operand scans the worktable to return the results.

2. **select max(c11) from t1**

```
( plan
    ( t_scan t1 )
    ( )
)
```

Returns a scalar aggregate. The last derived table is empty, because scalar aggregates accumulate the result value in an internal variable rather than a worktable.

3. **select ***
   from t1
   where c11 = (select count(*) from t2)

```
( plan
    ( i_scan i_c21 (table t2 ( in_subq 1) ) )
    ( i_scan i_c11 t1 )
)
```

Specifies the execution of a materialized subquery.

4. **create view v3**
   as
   select distinct * from t3

   select * from t1, v3
   where c11 = c31

```
( plan
    ( store Worktab1
        ( t_scan (table t3 (in_view v3 ) ) )
    )
    ( nl_g_join
        ( t_scan t1 )
        ( t_scan ( work_t Worktab1 ) )
    )
)
```

Specifies the execution of a materialized view.

Comments

- The **plan** operator is required for multistep queries, including:
  - Queries that generate worktables, such as queries that perform sorts and those that compute vector aggregates
  - Queries that compute scalar aggregates
  - Queries that include materialized subqueries

- An abstract plan for a query that requires multiple execution steps must include operands for each step in query execution if it begins with the **plan** keyword. Use the **hints** operator to introduce partial plans.

See Also

hints

prefetch

Function

Specifies the I/O size to use for the scan of a stored table.

Syntax

```
( prop table_name
       ( prefetch size )
   )
```

Operands

table_name – is the table to which the property is to be applied.

size- is a valid I/O size: 2, 4, 8 or 16.

Side Effects

The specified I/O size is used in the resultant query plan if a pool of that size exists in the cache used by the table.

Examples

1. **select * from t1**

```
( prop t1
    (prefetch 16 )
)
```

16K I/O size is used for the scan of *t1*.

Comments

- Partial plans can specify scan properties without specifying other portions of the query plan.
- If large I/O specifications in a saved plan do not match current pool configuration or other options:
  - If the plan specifies 16K I/O, and the 16K pool does not exist, the next largest available I/O size is used.
  - If session or server-level options have made large I/O unavailable for the query (**set prefetch** for the session, or **sp_cachestrategy** for the table), 2K I/O is used.
- If you save plans that specify only 2K I/O for the scan properties, and later create large I/O pools, enable **replace** mode to save the new plans if you want these plans to use larger I/O sizes.

See Also

prop

prop

Function

Specifies properties to use for the scan of a stored table.

Syntax

```
( prop table_name
      ( property_specification ) ...
   )
```

property_specification:

```
      ( prefetch size )
      ( lru | mru )
      ( parallel degree )
```

Operands

table_name – is the table to which the property is to be applied.

Side Effects

The specified properties are used for the scan of the table.

Examples

1. **select * from t1**

```
( t_scan t1 )
( prop t1
    ( parallel 1 )
    ( prefetch 16 )
    ( lru )
)
```

Shows the property values used by the scan of *t1*.

Comments

- Partial plans can specify scan properties without specifying other portions of the query plan.
- Generated plans include the parallel, prefetch, and cache strategy properties used for each table in the query.

See Also

lru, **mru**, **parallel**, **prefetch**

scan

Function

Specifies the scan of a stored table, without specifying the type of scan.

Syntax

(scan *stored_table*)

Operands

stored_table – is the name of the stored table to be scanned. It can be a base table or worktable.

Result

A derived table produced by the scan of the stored table.

Side Effects

The optimizer chooses the access method for the stored table.

Examples

1. **select * from t1 where c11 > 10**

```
( scan t1 )
```

Specifies a scan of *t1*, leaving the optimizer to choose whether to perform a table scan or index scan.

2. **select ***

```
from t1, t2
where c11 = 0
    and c22 = 1000
    and c12 = c21
```

```
( nl_g_join
        ( scan t2 )
        ( i_scan i_c22 t1 )
)
```

Specifies a partial plan, indicating the join order, but allowing the optimizer to choose the access method for *t2*.

Comments

- The **scan** operator is used when the choice of the type of scan should be left to the optimizer. The resulting access method can be one of the following:
  - A full table scan
  - An index scan, with access to data pages
  - A covering index scan, with no access to data pages
  - A RID scan, used for the OR strategy
- For an example of an abstract plan that specifies the reformatting strategy, see **store**.

See Also

i_scan, **store**, **t_scan**

store

Function

Stores the results of a scan in a worktable.

Syntax

```
( store worktable_name
      ( [scan | i_scan | t_scan ] table_name )
   )
```

Operands

worktable_name – is the name of the worktable to be created.

table_name – is the name of the base table to be scanned.

Result

A worktable that is the result of the scan.

Side Effects

The specified table is scanned, and the result is stored in a worktable.

Examples

1. **select c12, max(c11) from t1 group by c12**

```
( plan
    ( store Worktab1
        ( t_scan t1 )
    )
    ( t_scan ( work_t Worktab1 ) )
)
```

Specifies the two-step process of selecting the vector aggregate into a worktable, then selecting the results of the worktable.

Comments

- The legal places for a **store** operator in an abstract plan are:
  - Under a **plan** or **union** operator, where the **store** operator signifies a preprocessing step resulting in a worktable
  - Under a **scan** operator (but not under an **i_scan** or **t_scan** operator)

- During plan capture mode, worktables are identified as *Worktab1*, *Worktab2*, and so on. For manually entered plans, any naming convention can be used.
- The use of the reformatting strategy can be described in an abstract plan using the **scan** (**store** ( )) combination of operators. For example, if *t2* has no indexes and is very large, the abstract plan below indicates that *t2* should be scanned once, via a table scan, with the results stored in a worktable:

```
select *
from t1, t2
where c11 > 0
        and c12 = c21
        and c22 between 0 and 10000
```

```
( nl_g_join
        (i_scan i_c11 t1)
        ( scan (store (t_scan t2 )))
)
```

See Also

scan

subq

Function

Identifies a subquery.

Syntax

```
( subq subquery_id
        )
```

Operands

subquery_id – is an integer identifying the subquery. In abstract plans, subquery numbering is based on the order of the leading parenthesis for the subqueries in a query.

Examples

1. **select c11 from t1**
   where c12 =
   (select c21 from t2 where c22 = t1.c11)

```
( nested
        ( t_scan t1 )
        ( subq 1
                ( t_scan ( table t2 ( in ( subq 1 ) ) ) )
        )
)
```

A single nested subquery.

2. **select c11 from t1**
   where c12 =
   (select c21 from t2 where c22 = t1.c11)
   and c12 =
   (select c31 from t3 where c32 = t1.c11)

```
( nested
        ( nested
                ( t_scan t1 )
                ( subq 1
                        ( t_scan ( table t2 ( in ( subq 1 ) ) ) )
                )
        )
        ( subq 2
                ( t_scan ( table t3 ( in ( subq 2 ) ) ) )
        )
)
```

The two subqueries are both nested in the main query.

3. **select c11 from t1
where c12 =
(select c21 from t2 where c22 =
(select c31 from t3 where c32 = t1.c11))**

```
( nested
      ( t_scan t1 )
      ( subq 1
            ( nested
                 ( t_scan ( table t2 ( in ( subq 1 ) ) ) )
                 ( subq 2
                      ( t_scan ( table t3 ( in ( subq 2 ) ) ) )
                 )
            )
      )
)
```

A level 2 subquery nested into a level 1 subquery nested in the main query.

Comments

- The **subq** operator has two meanings in an abstract plan expression:
  - Under a **nested** operator, it describes the attachment of a nested subquery to a table
  - Under an **in** operator, it describes the nesting of the base tables and views that the subquery contains
- To specify the attachment of a subquery without providing a plan specification, use an empty hint:

```
( nested
    ( t_scan t1)
    ( subq 1
            ()
    )
)
```

- To provide a description of the abstract plan for a subquery, without specifying its attachment, specify an empty hint as the derived table in the **nested** operator:

```
( nested
    ()
    ( subq 1
        (t_scan ( table t1 ( in ( subq 1 ) ) ) )
    )
)
```

- When subqueries are flattened to a join, the only reference to the subquery in the abstract plan is the identification of the table specified in the subquery:

```
select *
from t2
where c21 in (select c12 from t1)
```

```
( nl_g_join
    ( t_scan t1 )
    ( t_scan ( table t2 ( in ( subq 1 ) ) ) )
```

- When a subquery is materialized, the subquery appears in the **store** operation, identifying the table to be scanned during the materialization step:

```
select *
from t1
where c11 in (select max(c22) from t2 group by c21)
```

```
( plan
    ( store Worktab1
        ( t_scan ( table t2 ( in ( subq 1 ) ) ) )
    )
    ( nl_g_join
        ( t_scan t1 )
        ( t_scan ( work_t Worktab1 ) )
    )
)
```

See Also

in, **nested**, **table**

t_scan

Function

Specifies a table scan of a stored table.

Syntax

(t_scan *stored_table*)

Operands

stored_table – is the name of the stored table to be scanned.

Result

A derived table produced by the scan of the stored table.

Side Effects

Instructs the optimizer to perform a table scan on the stored table.

Examples

1. **select * from t1**

   (t_scan t1)

   Performs a table scan of *t1*.

Comments

- Specifying **t_scan** forbids the use of reformatting and the OR strategy.

See Also

i_scan, scan, store

table

Function

Identifies a base table that occurs in a subquery or view or that is assigned a correlation name in the **from** clause of the query.

Syntax

(table *table_name* [*qualification*])

(table (*correlation_name table_name*))

Operands

table_name – is a base table. If the query uses the database name and / or owner name, the abstract plan must also provide them.

correlation_name –is the correlation name, if a correlation name is used in the query.

qualification – is either **in (subq** *subquery_id***)** or **in (view** *view_name***)**.

Side Effects

The specified derived tables in the abstract plan are matched against the positionally corresponding tables specified in the query.

Examples

1. **select * from t1 table1, t2 table2**
   where table1.c11 = table2.c21

```
( nl_g_join
    ( t_scan ( table ( table1 t1 ) ) )
    ( t_scan ( table ( table2 t2 ) ) )
)
```

   Tables *t1* and t2 are identified by reference to the correlation names used in the query.

2. **select c11 from t1**
   where c12 =
   (select c21 from t2 where c22 = t1.c11)

```
( nested
        ( t_scan t1 )
        ( subq 1
                ( t_scan ( table t2 ( in ( subq 1 ) ) ) )
        )
)
```

Table *t2* in the subquery is identified by reference to the subquery.

3.
```
create view v1
as
select * from t1 where c12 > 100

select t1.c11 from t1, v1
where t1.c12 = v1.c11
```

```
( nl_g_join
        ( t_scan t1 )
        ( i_scan 2 ( table t1 ( in ( view v1 ) ) )
)
```

Table *t1* in the view is identified by reference to the view.

Comments

- The **table** operator is used to link table names in an abstract plan to the corresponding table in a SQL query in queries that contain views, subqueries, and correlation names for tables.
- When correlation names are used, all references to the table, including those in the scan properties section, are in the form:

  (table (*correlation_name table_name*))

  The **table** operator is used for all references to the table, including the scan properties for the table under the **props** operator.

See Also

in, **subq**, **view**

union

Function

Describes the union of the two or more derived tables.

Syntax

```
(union
       derived_table1
       ...
       derived_tableN
)
```

Operands

derived_table1...*derived_tableN* – is the derived tables to be united.

Result

A derived table that is the union of the specified operands.

Side Effects

The specified derived tables in the abstract plan are matched against the positionally corresponding tables specified in the query.

Examples

1.
```
select * from t1
union
select * from t2
union
select * from t3
```

```
(union
    (t_scan t1)
    (t_scan t2)
    (t_scan t3)
)
```

Returns the union of the three full table scans.

2. **select 1,2**
   union
   select * from t2

```
(union
    ( )
    (tscan t2)
)
```

Since the first side of the union is not an optimizable query, the first union operand is empty.

Comments

- The **union** operator describes the processing for:
  - **union**, which removes duplicate values and
  - **union all**, which preserves duplicate values
- The **union** operator in an abstract query plan must have the same number of union sides as the SQL query and the order of the operands for the abstract plan must match the order of tables in the query.
- The sort step and worktable required to process **union** queries are not represented in abstract plans.
- If union queries list nonoptimizable elements, an empty operand is required. A **select** query that has no **from** clause is shown in example 2.

See Also

i_scan, **scan**, **t_scan**

view

Function

Identifies a view that contains the base table to be scanned.

Syntax

```
view view_name
```

Operands

view_name – is the name of a view specified in the query. If the query uses the database name and / or owner name, the abstract plan must also provides them.

Examples

1.
```
create view v1 as
select * from t1

select * from v1
```

```
( t_scan ( table t1 ( in ( view v ) ) ) )
```

Identifies the view in which table *t1* is used.

Comments

- When a query includes a view, the table must be identified using **table (***tablename* (**in** *view_name*)).

See Also

in, **table**

work_t

Function

Describes a stored worktable.

Syntax

```
( work_t [ worktable_name
        | (correlation_name worktable_name) ]
   )
```

Operands

worktable_name – is the name of a worktable.

correlation_name – is the correlation name specified for a worktable, if any.

Result

A stored table.

Side Effects

Matches the stored table against a work table in the query plan.

Examples

1. **select c12, max(c11) from t1**
   group by c12

```
( plan
    ( store Worktab1
        ( t_scan t1 )
    )
    ( t_scan ( work_t Worktab1 ) )
)
```

Specifies the two-step process of selecting vector aggregates into a worktable, then selecting the results of the worktable.

Comments

- The **store** operator creates a worktable; the **work_t** operator identifies a stored worktable for later access in the abstract plan.
- During plan capture mode, worktables are identified as *Worktab1*, *Worktab2*, and so on. For manually entered plans, any naming convention can be used.

- If the scan of the worktable is never specified explicitly with a scan operator, the worktable does not have to be named and the **work_t** operator can be omitted. The following plan uses an empty scan operator "( )" in place of the **t_scan** and **work_t** specifications used in example 1:

```
( plan
    ( store
        ( t_scan titles )
        )
    ()
)
```

- Correlation names for worktables are needed only for self-joined materialized views, for example:

```
create view v
as
select distinct c11 from t1

select *
from v v1, v v2
where ...

( plan
    ( store Worktab1
        ( t_scan ( table t1 (in ( view v ) ) ) )
    )
    ( g_join
        ( t_scan (work_t ( v1 Worktab1 ) ) )
        ( t_scan (work_t ( v2 Worktab1 ) ) )
    )
)
```

See Also

store, **view**

25 A User's Guide to Abstract Query Plans

This chapter contains the following sections:

Introduction to Writing Abstract Plans

Abstract plans allow you to specify the desired execution plan of a query. Abstract plans provide an alternative to the session-level and query level options that force a join order, or specify the index, I/O size, or other query execution options. The session-level and query-level options are described in Chapter 22, "Creating and Using Abstract Plans."

There are several optimization decisions that cannot be specified with **set** commands or clauses included in the query text. Some examples are:

- Subquery attachment
- The join order for flattened subqueries
- Reformatting

In many cases, including **set** commands or changing the query text is not always possible or desired. Abstract plans provide an alternative, more complete method of influencing optimizer decisions.

Abstract plans are relational algebra expressions that are not included in the query text. They are stored in a system catalog and associated to incoming queries based on the text of these queries.

The tables used in this section are the same as those in Chapter 24, "Abstract Plan Language Reference." See "Schema for Examples" on page 24-2 for the **create table** and **create index** statements.

The Abstract Plan Language

The abstract plan language is a relational algebra that uses these operators:

- **g_join**, the generic join, a high-level logical join operator. It describes inner, outer and existence joins, using either nested-loop joins or sort-merge joins.
- **nl_g_join**, specifying a nested-loop join, including all inner, outer, and existence joins
- **m_g_join**, specifying a merge join, including inner and outer joins.
- **union**, a logical union operator. It describes both the **union** and the **union all** SQL constructs.
- **scan**, a logical operator that transforms a stored table in a flow of rows, a **derived table**. It allows partial plans that do not restrict the access method.
- **i_scan**, a physical operator, implementing **scan**. It directs the optimizer to use an index scan on the specified table.
- **t_scan**, a physical operator, implementing **scan**. It directs the optimizer to use a full table scan on the specified table.
- **store**, a logical operator, describing the materialization of a derived table in a stored worktable.
- **nested**, a filter, describing the placement and structure of nested subqueries.

See “Schema for Examples” on page 24-2 for the **create table** and **create index** commands used for the examples in this section.

Additional abstract plan keywords are used for grouping and identification:

- **plan** groups the elements when a plan requires multiple steps.
- **hints** groups a set of hints for a partial plan.
- **prop** introduces a set of scan properties for a table: **prefetch**, **lru|mru** and **parallel**.
- **table** identifies a table when correlation names or tables in subqueries or views.
- **work_t** identifies a worktable.
- **in**, used with **table**, for identifying tables named in a subquery (**subq**) or view (**view**).

- **subq** is also used under the nested operator to indicate the attachment point for a nested subquery, and to introduce the subqueries abstract plan.

Queries, Access Methods, and Abstract Plans

For any specific table, there can be several access methods for a specific query: index scans using different indexes, table scans, the OR strategy, and reformatting are some examples.

This simple query has several choices of access methods:

```
select * from t1
where c11 > 1000 and c12 < 0
```

The following abstract plans specify three different access methods:

- Use the index *i_c11*:

  ```
  (i_scan i_c11 t1)
  ```

- Use the index *i_c12*:

  ```
  (i_scan i_c12 t1)
  ```

- Do a full table scan:

  ```
  (t_scan t1)
  ```

Abstract plans can be full plans, specifying all optimizer choices for a query, or can specify a subset of the choices, such as the index to use for a single table in the query, but not the join order for the tables. For example, using a partial abstract plan, you can specify that the query above should use some index and let the optimizer choose between *i_c11* and *i_c12*, but not do a full table scan. The empty parentheses are used in place of the index name:

```
(i_scan () t1)
```

In addition, the query could use either 2K or 16K I/O, or be performed in serial or parallel.

Identifying Tables in Abstract Plans

Abstract plans need to name all of a query's tables in a non-ambiguous way, such that a table named in the abstract can be linked to its occurrence in the SQL query. In most cases, the table name is all that is needed. If the query qualifies the table name with the database and owner name, these are also needed to fully identify a table in the abstract plan. For example, this example used the unqualified table name:

```
select * from t1
```

The abstract plan also uses the unqualified name:

```
(t_scan t1)
```

If a database name and / or owner name are provided in the query:

```
select * from pubs2.dbo.t1
```

Then the abstract plan must also use the qualifications:

```
(t_scan pubs2.dbo.t1)
```

However, the same table may occur several times in the same query, as in this example:

```
select * from t1 a, t1 b
```

Correlation names, *a* and *b* in the example above, identify the two tables in SQL. In an abstract plan, the **table** operator associates each correlation name with the occurrence of the table:

```
( g_join
        ( t_scan ( table ( a t1 ) ) )
        ( t_scan ( table ( b t1 ) ) )
)
```

Table names can also be ambiguous in views and subqueries, so the **table** operator is used for tables in views and subqueries.

For subqueries, the **in** and **subq** operators qualify the name of the table with its syntactical containment by the subquery. The same table is used in the outer query and the subquery in this example:

```
select *
from t1
where c11 in (select c12 from t1 where c11 > 100)
```

The abstract plan identifies them unambiguously:

```
( g_join
     ( t_scan t1 )
     ( i_scan i_c11_c12 ( table t1 ( in ( subq 1 ) ) ) )
)
```

For views, the **in** and **view** operators provide the identification. The query in this example references a table used in the view:

```
create view v1
as
select * from t1 where c12 > 100

select t1.c11 from t1, v1
    where t1.c12 = v1.c11
```

Here is the abstract plan:

```
( g_join
    ( t_scan t1 )
    ( i_scan i_c12 ( table t1 ( in ( view v1 ) ) ) )
)
```

A table may be deeply nested, for instance in a view of a subquery of a view of a view of a view of a subquery of the main query. The table name is accordingly (and recursively) qualified with the whole chain of syntactic containment. See example 3 on page 24-12.

Identifying Indexes in Abstract Plans

The **i_scan** operator requires two operands, the index name and the table name, as shown here:

```
( i_scan i_c12 t1 )
```

To specify that some index should be used, without specifying the index, substitute empty parenthesis for the index name:

```
( i_scan ( ) t1 )
```

Specifying Join Order in Abstract Plans

Adaptive Server performs joins of three or more tables by joining two of the tables, and joining the "derived table" from that join to the next table in the join order. This derived table is a flow of rows, as from an earlier nested-loop join in the query execution.

This query joins three tables:

```
select *
from t1, t2, t3
where c11 = c21
    and c12 = c31
    and c22 = 0
    and c32 = 100
```

This example shows the binary nature of the join algorithm, using **g_join** operators. The plan specifies the join order *t2*, *t1*, *t3*:

```
(g_join
    (g_join
        (scan t2)
        (scan t1)
    )
    (scan t3)
)
```

The results of the *t2*-*t1* join are then joined to *t3*. The **scan** operator in this example leaves the choice of table scan or index scan up to the optimizer.

A Shorthand Notation for Joins

In general, a *N*-way join, with the order *t1*, *t2*, *t3*..., *tN*-1, *tN* is described by:

```
(g_join
    (g_join
        ...
            (g_join
                (g_join
                    (scan t1)
                    (scan t2)
                )
                (scan t3)
            )
        ...
        (scan tN-1)
    )
    (scan tN)
)
```

This notation can be used as shorthand for the **g_join** operator:

```
(g_join
    (scan t1)
    (scan t2)
    (scan t3)
    ...
    (scan tN-1)
    (scan tN)
)
```

This notation can be used for **g_join**, and **nl_g_join**, and **m_g_join**.

Join Order Examples

The optimizer could select among several plans for this three-way join query:

```
select *
from t1, t2, t3
where c11 = c21
    and c12 = c31
    and c22 = 0
    and c32 = 100
```

Here are a few examples:

- Use *c22* as a search argument on *t2*, join with *t1* on *c11*, then with *t3* on *c31*:

```
(g_join
    (i_scan i_c22 t2)
    (i_scan i_c11 t1)
    (i_scan i_c31 t3)
)
```

- Use the search argument on *t3*, and the join order *t3*, *t1*, *t2*:

```
(g_join
    (i_scan i_c32 t3)
    (i_scan i_c12 t1)
    (i_scan i_c21 t2)
)
```

- Do a full table scan of *t2*, if it is small and fits in cache, still using the join order *t3*, *t1*, *t2*:

```
(g_join
    (i_scan i_c32 t3)
    (i_scan i_c12 t1)
    (t_scan t2)
)
```

- If *t1* is very large, and *t2* and *t3* individually qualify a large part of *t1*, but together a very small part, this plan specifies a STAR join:

```
(g_join
    (i_scan i_c22 t2)
    (i_scan i_c32 t3)
    (i_scan i_c11_c12 t1)
)
```

All of these plans completely constrain the choice of join order, letting the optimizer choose the type of join.

The generic **g_join** operator implements outer joins, inner joins, and existence joins. For examples of flattened subqueries that perform existence joins, see “Flattened Subqueries” on page 25-13.

Match Between Execution Methods and Abstract Plans

There are some limits to join orders and join types, depending on the type of query. One example is outer joins, such as:

```
select * from t1, t2
where c11 *= c21
```

Adaptive Server requires the outer member of the outer join to be the outer table during join processing. Therefore, this abstract plan is illegal:

```
(g_join
    (scan t2)
    (scan t1)
)
```

Attempting to use this plan results in an error message, and the query is not compiled.

Specifying Join Order for Queries Using Views

You can use abstract plans to enforce the join order for merged views. This example creates a view. This view performs a join of *t2* and *t3*:

```
create view v2
as
select *
from t2, t3
where c22 = c32
```

This query performs a join with the *t2* in the view:

```
select * from t1, v2
where c11 = c21
    and c22 = 0
```

This abstract plan specifies the join order *t2*, *t1*, *t3*:

```
(g_join
    (scan (table t2 (in (view v2))))
    (scan t1)
    (scan (table t3 (in (view v2))))
)
```

This example joins with *t3* in the view:

```
select * from t1, v2
where c11 = c31
    and c32 = 100
```

This plan uses the join order *t3*, *t1*, *t2*:

```
(g_join
    (scan (table t3 (in (view v2))))
    (scan t1)
    (scan (table t2 (in (view v2))))
)
```

This is an example where abstract plans can be used, if needed, to affect the join order for a query, when **set forceplan** cannot.

Specifying the Join Type in Abstract Plans

Adaptive Server can perform either nested-loop or merge joins. The **g_join** operator leaves the optimizer free to choose the best join algorithm, based on costing. To specify a nested-loop join, use the **nl_g_join** operator; for a sort-merge join, use the **m_g_join** operator. Abstract plans captured by Adaptive Server always include the operator that specifies the algorithm, and not the **g_join** operator.

Note that the “g” that appears in each operator means “generic,” meaning that they apply to inner joins and outer joins; **g_join** and **nl_g_join** can also apply to existence joins.

This query specifies a join between *t1* and *t2*:

```
select * from t1, t2
    where c12 = c21 and c11 = 0
```

This abstract plan specifies a nested-loop join:

```
(nl_g_join
    (i_scan i_c11 t1)
    (i_scan i_c21 t2)
)
```

The nested-loop plan uses the index *i_c11* to limit the scan using the search clause, and then performs the join with *t2*, using the index on the join column.

This merge-join plan uses different indexes:

```
(m_g_join
    (i_scan i_c12 t1)
    (i_scan i_c21 t2)
)
```

The merge join uses the indexes on the join columns, *i_c12* and *i_c21*, for the merge keys. This query performs a full-merge join and no sort is needed.

A merge join could also use the index on *i_c11* to select the rows from *t1* into a worktable; the merge uses the index on *i_c21*:

```
(m_g_join
    (i_scan i11 t1)
    (i_scan i21 t2)
)
```

The step that creates the worktable is not specified in the plan; the optimizer detects when a worktable and sort are needed for join-key ordering.

Specifying Partial Plans and Hints

There are cases when a full plan is not needed. For example, if the only problem with a query plan is that the optimizer chooses a table scan instead of using a nonclustered index, the abstract plan can specify only the index choice, and leave the other decisions to the optimizer.

The optimizer could choose a table scan of *t3* rather than using *i_c31* for this query:

```
select *
from t1, t2, t3
where c11 = c21
    and c12 < c31
    and c22 = 0
    and c32 = 100
```

The following plan, as generated by the optimizer, specifies join order *t2*, *t1*, *t3*. However, the plan specifies a table scan of *t3*:

```
(g_join
    (i_scan i_c22 t2)
    (i_scan i_c11 t1)
    (t_scan t3)
)
```

This full plan could be modified to specify the use of *i_c31* instead:

```
(g_join
    (i_scan i_c22 t2)
    (i_scan i_c11 t1)
    (i_scan i_c31 t3)
)
```

However, specifying only a partial abstract plan is a more flexible solution. As data in the other tables of that query evolves, the optimal join order can change. The partial plan can specify just one partial plan item. For the index scan of *t3*, the partial plan is simply:

```
(i_scan i_c31 t3)
```

The optimizer chooses the join order and the access methods for *t1* and *t2*.

Grouping Multiple Hints

There may be cases where more than one plan fragment is needed. For example, you might want to specify that some index should be used for each table in the query, but leave the join order up to the optimizer. When multiple hints are needed, they can be grouped with the **hints** operator:

```
(hints
    (i_scan () t1)
    (i_scan () t2)
    (i_scan () t3)
)
```

In this case, the role of the **hints** operator is purely syntactic; it does not affect the ordering of the scans.

There are no limits on what may be given as a hint. Partial join orders may be mixed with partial access methods. This hint specifies that *t2* is outer to *t1* in the join order, and that the scan of *t3* should use an index, but the optimizer can choose the index for *t3*, the access methods for *t1* and *t2*, and the placement of *t3* in the join order:

```
(hints
    (g_join
        (scan t2)
        (scan t1)
    )
    (i_scan () t3)
)
```

Inconsistent and Illegal Plans Using Hints

It is possible to describe inconsistent plans using hints, such as this plan that specifies contradictory join orders:

```
(hints
    (g_join
        (scan t2)
        (scan t1)
    )
    (g_join
        (scan t1)
        (scan t2)
    )
)
```

When the query associated with the plan is executed, the query cannot be compiled, and an error is raised.

Other inconsistent hints do not raise an exception, but may use any of the specified access methods. This plan specifies both an index scan and a table scan for the same table:

```
(hints
    (t_scan t3)
    (i_scan () t3)
)
```

In this case, either method may be chosen, the behavior is indeterminate.

Creating Abstract Plans for Subqueries

Subqueries are resolved in several ways in Adaptive Server, and the abstract plans reflect the query execution steps:

- Materialization – The subquery is executed and results are stored in a worktable or internal variable. See “Materialized Subqueries” on page 25-12.
- Flattening – The query is flattened into a join with the tables in the main query. See “Flattened Subqueries” on page 25-13.
- Nesting – The subquery is executed once for each outer query row. See “Nested Subqueries” on page 25-15.

Abstract plans do not allow the choice of the basic subquery resolution method. This is a rule-based decision and cannot be changed during query optimization. Abstract plans, however, can be used to influence the plans for the outer and inner queries. In nested subqueries, abstract plans can also be used to choose where the subquery is nested in the outer query.

Materialized Subqueries

This query includes a noncorrelated subquery that can be materialized:

```
select *
from t1
where c11 = (select count(*) from t2)
```

The first step in the abstract plan materializes the scalar aggregate in the subquery. The second step uses the result to scan *t1*:

```
( plan
     ( i_scan i_c21 ( table t2 ( in (subq 1 ) ) ) )
     ( i_scan i_c11 t1 )
)
```

This query includes a vector aggregate in the subquery:

```
select *
from t1
where c11 in (select max(c21)
        from t2
        group by c22)
```

The abstract plan materializes the subquery in the first step, and joins it to the outer query in the second step:

```
( plan
    ( store Worktab1
        ( t_scan ( table t2 ( in (subq 1 ) ) ) )
    )
    ( nl_g_join
        ( t_scan t1 )
        ( t_scan ( work_t Worktab1 ) )
    )
)
```

Flattened Subqueries

Some subqueries can be flattened into joins. The **g_join** and **nl_g_join** operators leave it to the optimizer to detect when an existence join is needed. For example, this query includes a subquery introduced with exists:

```
select * from t1
where c12 > 0
    and exists (select * from t2
            where t1.c11 = c21
                and c22 < 100)
```

The semantics of the query require an existence join between *t1* and *t2*. The join order *t1*, *t2* is interpreted by the optimizer as an existence join, with the scan of *t2* stopping on the first matching row of *t2* for each qualifying row in *t1*:

```
(g_join
    (scan t1)
    (scan (table t2 (in (subq 1) ) ))
)
```

The join order *t2*, *t1* requires other means to guarantee the duplicate elimination:

```
(g_join
    (scan (table t2 (in (subq 1) ) ) )
    (scan t1)
)
```

Using this abstract plan, the optimizer can decide to use:

- A unique index on *t2.c21*, if one exists, with a regular join.
- The unique reformatting strategy, if no unique index exists. In this case, the query will probably use the index on *c22* to select the rows into a worktable.
- The duplicate elimination sort optimization strategy, performing a regular join and selecting the results into the worktable, then sorting the worktable.

The abstract plan does not need to specify the creation and scanning of the worktables needed for the last two options.

For more information on subquery flattening, see “Flattening in, any, and exists Subqueries” on page 7-23.

Example: Changing the Join Order in a Flattened Subquery

The query can be flattened to an existence join:

```
select *
from t1, t2
where c11 = c21
    and c21 > 100
    and exists (select * from t3
        where c31 != t1.c11)
```

The “!=” correlation can make the scan of *t3* rather expensive. If the join order is *t1*, *t2*, the best place for *t3* in the join order depends on whether the join of *t1* and *t2* increases or decreases the number of rows, and therefore, the number of times that the expensive table scan needs to be performed. If the optimizer fails to find the right join order for *t3*, the following abstract plan can be used when the join reduces the number of times that *t3* must be scanned:

```
(g_join
    (scan t1)
    (scan t2)
    (scan (table t3 (in (subq 1) ) ) )
)
```

If the join increases the number of times that *t3* needs to be scanned, this abstract plan performs the scans of *t3* before the join:

```
(g_join
    (scan t1)
    (scan (table t3 (in (subq 1) ) ) )
    (scan t2)
)
```

Nested Subqueries

Nested subqueries can be explicitly described in abstract plans:

- The abstract plan for the subquery is provided.
- The location at which the subquery attaches to the main query is specified.

Abstract plans allow you to affect the query plan for the subquery, and to change the attachment point for the subquery in the outer query.

The **nested** operator specifies the position of the subquery in the outer query. Subqueries are "nested over" a specific derived table. The optimizer chooses a spot where all the correlation columns for the outer query are available, and where it estimates that the subquery needs to be executed the least number of times.

The following SQL statement contains a correlated expression subquery:

```
select *
from t1, t2
where c11 = c21
    and c21 > 100
    and c12 = (select c31 from t3
                where c32 = t1.c11)
```

The abstract plan shows the subquery nested over the scan of *t1*:

```
( g_join
     ( nested
          ( i_scan i_c12 t1 )
          ( subq 1
              (t_scan ( table t3 ( in ( subq 1 ) ) ) )
           )
     )
     ( i_scan i_c21 t2 )
)
```

Subquery Identification and Attachment

Subqueries are identified with numbers, in the order of their leading opened parenthesis "(".

This example has two subqueries, one in the select list:

```
select
    (select c11 from t1 where c12 = t3.c32), c31
from t3
where c32 > (select c22 from t2 where c21 = t3.c31)
```

In the abstract plan, the subquery containing *t1* is named "1" and the subquery containing *t2* is named "2". Both subquery 1 and 2 are nested over the scan of *t3*:

```
( nested
    ( nested
        ( t_scan t3 )
        ( subq 1
            ( i_scan i_c11_c12 ( table t1 (in ( subq 1 ) ) ) )
        )
    )
    ( subq 2
        ( i_scan i_c21 ( table t2 ( in ( subq 2 ) ) ) )
    )
)
```

In this query, the second subquery is nested in the first:

```
select * from t3
where c32 > all
    (select c11 from t1 where c12 > all
        (select c22 from t2 where c21 = t3.c31))
```

In this case, the subquery containing *t1* is also named "1" and the subquery containing *t2* is named "2". In this plan, subquery 2 is nested over the scan of *t1*, which is performed in subquery 1; subquery 1 is nested over the scan of *t3* in the main query:

```
( nested
    ( t_scan t3 )
    ( subq 1
        ( nested
            ( i_scan i_c11_c12 ( table t1 ( in ( subq 1 ) ) ) )
            ( subq 2
                ( i_scan i_c21 ( table t2 ( in ( subq 2 ) ) ) )
            )
        )
    )
```

More Subquery Examples: Reading Ordering and Attachment

The **nested** operator has the derived table as the first operand and the nested subquery as the second operand. This allows an easy vertical reading of the join order and subquery placement:

```
select *
from t1, t2, t3
where c12 = 0
    and c11 = c21
    and c22 = c32
    and 0 < (select c21 from t2 where c22 = t1.c11)
```

In the plan, the join order is *t1*, *t2*, *t3*, with the subquery nested over the scan of *t1*:

```
( g_join
    ( nested
        ( i_scan i_c11 t1 )
        ( subq 1
            ( t_scan ( table t2 ( in (subq 1 ) ) )
        )
    )
    ( i_scan i_c21 t2 )
    ( i_scan i_c32 t3 )
)
```

Modifying Subquery Nesting

If you modify the attachment point for a subquery, you must choose a point at which all of the correlation columns are available.This query is correlated to both of the tables in the outer query:

```
select *
from t1, t2, t3
where c12 = 0
    and c11 = c21
    and c22 = c32
    and 0 < (select c31 from t3 where c31 = t1.c11
                    and c32 = t2.c22)
```

This plan uses the join order *t1*, *t2*, *t3*, with the subquery nested over the *t1*-*t2* join:

```
( g_join
    ( nested
        ( g_join
            ( i_scan i_c11_c12 t1 )
            ( i_scan i_c22 t2 )
        )
        ( subq 1
            ( t_scan ( table t3 ( in (subq 1 ) ) )
)
        )
    )
    ( i_scan i_c32 t3 )
)
```

Since the subquery requires columns from both outer tables, it would be incorrect to nest it over the scan of *t1* or the scan of *t2*; such errors are silently corrected during optimization.

Abstract Plans for Materialized Views

This view is materialized during query processing:

```
create view v3
as
select distinct *
from t3
```

This query performs a join with the materialized view:

```
select *
from t1, v3
where c11 = c31
```

A first step materializes the view *v3* into a worktable. The second joins it with the main query table *t1*:

```
( plan
        ( store Worktab1
                ( t_scan ( table t3 ( in (view v3 ) ) ) )
        )
        ( g_join
                ( t_scan t1 )
                ( t_scan ( work_t Worktab1 ) )
        )
)
```

Abstract Plans for Queries Containing Aggregates

This query returns a scalar aggregate:

```
select max(c11) from t1
```

The first step computes the scalar aggregate and stores it in an internal variable. The second step is empty, as it only returns the variable, in a step with nothing to optimize:

```
( plan
        ( t_scan t1 )
        ( )
)
```

Vector aggregates are also two-step queries:

```
select max(c11)
from t1
group by c12
```

The first step processes the aggregates into a worktable; the second step scans the worktable:

```
( plan
        ( store Worktab1
                ( t_scan t1 )
        )
        ( t_scan ( work_t Worktab1 ) )
)
```

Nested aggregates are a Transact-SQL extension:

```
select max(count(*))
from t1
group by c11
```

The first step processes the vector aggregate into a worktable, the second scans it to process the nested scalar aggregate into an internal variable, and the third step returns the value.

```
( plan
     ( store Worktab1
          ( i_scan i_c12 t1 )
     )
     ( t_scan ( work_t Worktab1 ) )
     ( )
)
```

Extended columns in aggregate queries are a Transact-SQL extension:

```
select max(c11), c11
from t1
group by c12
```

The first step processes the vector aggregate; the second one joins it back to the base table to process the extended columns:

```
( plan
     ( store Worktab1
          ( t_scan t1 )
     )
     ( g_join
          ( t_scan t1 )
          ( i_scan i_c11 ( work_t Worktab1 ) )
     )
)
```

This example contains an aggregate in a merged view:

```
create view v4
as
select max(c11) as c41, c12 as c42
from t1
group by c12

select * from t2, v4
where c21 = 0
    and c22 > c41
```

The first step processes the vector aggregate; the second joins it to the main query table:

```
( plan
    ( store Worktab1
        ( t_scan ( table t1 ( in (view v4 ) ) ) )
    )
    ( g_join
        ( i_scan i_c22 t2 )
        ( t_scan ( work_t Worktab1 ) )
    )
)
```

This example includes an aggregate that is processed using a materialized view:

```
create view v5
as
select distinct max(c11) as c51, c12 as c52
from t1
group by c12
```

```
select * from t2, v5
where c21 = 0
    and c22 > c51
```

The first step processes the vector aggregate into a worktable. The second step scans it into a second worktable to process the materialized view. The third step joins this second worktable in the main query:

```
( plan
    ( store Worktab1
        ( t_scan ( table t1 ( in (view v5 ) ) ) )
    )
    ( store Worktab2
        ( t_scan ( work_t Worktab1 ) )
    )
    ( g_join
        ( i_scan i_c22 t2 )
        ( t_scan ( work_t Worktab2 ) )
    )
)
```

Specifying the Reformatting Strategy

In this query, *t2* is very large, and has no index:

```
select *
from t1, t2
where c11 > 0
    and c12 = c21
    and c22 = 0
```

The abstract plan that specifies the reformatting strategy on *t2* is:

```
( g_join
    (t_scan t1
    (scan
        (store Worktab1
            (t_scan t2)
        )
    )
)
```

In the case of the reformatting strategy, the **store** operator is an operand of **scan**. This is the only case when the *store* operator is not the operand of a *plan* operator.

OR Strategy Limitation

The OR strategy has no matching abstract plan that describes the RID scan required to perform the final step. All abstract plans generated by Adaptive Server for the OR strategy specify only the **scan** operator. You cannot use abstract plans to influence index choice for queries that require the OR strategy to eliminate duplicates.

When the *store* Operator Is Not Specified

Some multistep queries that require worktables do not require multistep plans with a separate worktable step, and the use of the **store** operator to create the worktable. These are:

- The sort step of queries using **distinct**
- The worktables needed for merge joins
- Worktables needed for **union** queries
- The sort step, when a flattened subquery requires sort to remove duplicates

Tips on Writing Abstract Plans

Here are some additional tips for writing and using abstract plans:

- Look at the current plan for the query and at plans that use the same query execution steps as the plan you need to write. It is often easier to modify an existing plan than to write a full plan from scratch.
  - Capture the plan for the query.
  - Use **sp_help_qplan** to display the SQL text and plan.
  - Edit this output to generate a **create plan** command, or attach an edited plan to the SQL query using the **plan** clause.
- It is often best to specify partial plans for query tuning in cases where most optimizer decisions are appropriate, but only an index choice, for example, needs improvement. By using partial plans, the optimizer can choose other paths for other tables as the data in other tables changes.
- Once saved, abstract plans are static. Data volumes and distributions may change so that saved abstract plans are no longer optimal. Subsequent tuning changes made by adding indexes, partitioning a table, or adding buffer pools may mean

that some saved plans are not performing as well as possible under current conditions. Most of the time, you want to operate with a small number of abstract plans that solve specific problems. Perform periodic plan checks to verify that the saved plans are still better than the plan that the optimizer would choose.

Comparing Plans “Before” and “After”

Abstract query plans can be used to assess the impact of an Adaptive Server software upgrade or system tuning changes on your query plans. You need to save plans before the changes are made, perform the upgrade or tuning changes, and then save plans again and compare the plans. The basic set of steps is:

1. Enable server-wide capture mode by setting the configuration parameter **abstract plan dump** to 1. All plans are then captured in the default group, *ap_stdout*.
2. Allow enough time for the captured plans to represent most of the queries run on the system. You can check whether additional plans are being generated by checking whether the count of rows in the *ap_stdout* group in *sysqueryplans* is stable:

   ```
   select count(*) from sysqueryplans where gid = 2
   ```

3. Copy all plans from *ap_stdout* to *ap_stdin* (or some other group, if you do not want to use server-wide plan load mode), using **sp_copy_all_qplans**.
4. Drop all query plans from *ap_stdout*, using **sp_drop_all_qplans**.
5. Perform the upgrade or tuning changes.
6. Allow sufficient time for plans to be captured to *ap_stdout*.
7. Compare plans in *ap_stdout* and *ap_stdin*, using the **diff** mode parameter of **sp_cmp_all_qplans**. For example, this query compares all plans in *ap_stdout* and *ap_stdin*:

   ```
   sp_cmp_all_qplans ap_stdout, ap_stdin, diff
   ```

   This displays only information about the plans that are different in the two groups.

Effects of Enabling Server-Wide Capture Mode

When server-wide capture mode is enabled, plans for all optimizable queries are saved in all databases on the server. Some possible system administration impacts are:

- When plans are captured, the plan is saved in *sysqueryplans* and log records are generated. The amount of space required for the plans and log records depends on the size and complexity of the SQL statements and query plans. Check space in each database where users will be active. You may need to perform more frequent transaction log dumps, especially in the early stages of server-wide capture when many new plans are being generated.
- If users execute system procedures from the *master* database, and *installmaster* was loaded with server-wide plan capture enabled, then plans for the optimizable statements in system procedures are saved in *master..sysqueryplans*. This is also true for any user-defined procedures created while plan capture was enabled. You may want to provide a default database at login for all users, including System Administrators, if space in *master* is limited.
- The *sysqueryplans* table uses datarows locking to reduce lock contention. However, especially when a large number of new plans are being saved, there may be a slight impact on performance.
- While server-wide capture mode is enabled, using **bcp** saves query plans in the *master* database. If you perform **bcp** using a large number of tables or views, check *sysqueryplans* and the transaction log in *master*.

The Time and Space to Copy Plans

If you have a large number of query plans in *ap_stdout*, be sure there is sufficient space to copy them on the *system* segment before starting the copy. Use **sp_spaceused** to check the size of *sysqueryplans*, and **sp_helpsegment** to check the size of the system segment.

Copying plans also requires space in the transaction log.

sp_copy_all_qplans calls **sp_copy_qplan** for each plan in the group to be copied. If **sp_copy_all_qplans** fails at any time due to lack of space or other problems, any plans that were successfully copied remain in the target query plan group.

Abstract Plans for Stored Procedures

For abstract plans to be captured for the optimizable SQL statements in stored procedures:

- The procedures must be created while plan capture or plan association mode is enabled. (This saves the text of the procedure in *sysprocedures*.)
- The procedure must be executed with plan capture mode enabled, and the procedure must be read from disk, not from the procedure cache.

This sequence of steps captures the query text and abstract plans for all optimizable statements in the procedure:

```
set plan dump dev_plans on
go
create procedure myproc as ...
go
exec myproc
go
```

If the procedure is in cache, so that the plans for the procedure are not being captured, you can execute the procedure **with recompile**. Similarly, once a stored procedure has been executed using an abstract query plan, the plan in the procedure cache is used so that query plan association does not take place unless the procedure is read from disk.

Procedures and Plan Ownership

When plan capture mode is enabled, abstract plans for the optimizable statements in a stored procedure are saved with the user ID of the owner of the procedure.

During plan association mode, association for stored procedures is based on the user ID of the owner of the procedure, not the user who executes the procedure. This means that once an abstract query plan is created for a procedure, all users who have permission to execute the procedure use the same abstract plan.

Procedures with Variable Execution Paths and Optimization

Executing a stored procedure saves abstract plans for each optimizable statement, even if the stored procedure contains control-of-flow statements that can cause different statements to be run depending on parameters to the procedure or other conditions. If the query is run a second time with different parameters that use a different code path, plans for any optimizable statements were saved by the earlier execution, and the abstract plan for the statement is associated with the query.

However, abstract plans for procedures do not solve the problem with procedures with statements that are optimized differently depending on conditions or parameters. One example is a procedure where users provide the low and high values for a **between** clause, with a query such as:

```
select title_id
from titles
where price between @lo and @hi
```

Depending on the parameters, the best plan could either be index access or a table scan. For these procedures, the abstract plan may specify either access method, depending on the parameters when the procedure was first executed. For more information on optimization of procedures, see “Splitting Stored Procedures to Improve Costing” on page 5-30.

Ad Hoc Queries and Abstract Plans

Abstract plan capture saves the full text of the SQL statement and abstract plan association is based on the full text of the SQL query. If users submit ad hoc SQL statements, rather than using stored procedures or Embedded SQL, abstract plans are saved for each different combination of query clauses. This can result in a very large number of abstract plans.

If users check the price of a specific *title_id* using **select** statements, an abstract plan is saved for each statement. The following two queries each generate an abstract plan:

```
select price from titles where title_id = "T19245"

select price from titles where title_id = "T40007"
```

In addition, there is one plan for each user, that is, if several users check for the *title_id* “T40007”, a plan is save for each user ID.

If such queries are included in stored procedures, there are two benefits:

- Only only one abstract plan is saved, for example, for the query:

```
select price from titles where title_id = @title_id
```

- The plan is saved with the user ID of the user who owns the stored procedure, and abstract plan association is made based on the procedure owner's ID.

Using Embedded SQL, the only abstract plan is saved with the host variable:

```
select price from titles
where title_id = :host_var_id
```

Sybase® Adaptive Server™ Enterprise Performance and Tuning Guide: System Tuning

Adaptive Server Enterprise Version 12

Document ID: 36216-01-1200-02

Last Revised: October 1999

Principal author: Karen Paulsell

Contributing authors: Server Publications Group

Document ID: 36216-01-1200-02

This publication pertains to Adaptive Server Enterprise Version 12 of the Sybase database management software and to any subsequent version until otherwise indicated in new editions or technical notes. Information in this document is subject to change without notice. The software described herein is furnished under a license agreement, and it may be used or copied only in accordance with the terms of that agreement.

Document Orders

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

Customers in other countries with a U.S. license agreement may contact Customer Fulfillment via the above fax number. All other international customers should contact their Sybase subsidiary or local distributor.

Upgrades are provided only at regularly scheduled software release dates.

Sybase Trademarks

Restricted Rights

Table of Contents

About This Book

Locking

26. Locking in Adaptive Server

27. Locking Commands

28. Reporting on Locks

29. Locking Configuration and Tuning

Application Maintenance

30. Maintenance Activities and Performance

31. Setting Space Management Properties

System-Level Tuning

32. Memory Use and Performance

33. Controlling Physical Data Placement

34. Tuning Asynchronous Prefetch

35. *tempdb* Performance Issues

36. Networks and Performance

37. How Adaptive Server Uses Engines and CPUs

38. Distributing Engine Resources Between Tasks

39. Monitoring Performance with *sp_sysmon*

Index

List of Figures

List of Tables

About This Book

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

Audience

This manual is intended for:

- Sybase System Administrators
- Database designers
- Application developers

How to Use This Book

The *Performance and Tuning Guide* is divided into these volumes:

- *Performance and Tuning: Query Tuning*
- *Performance and Tuning: System Tuning*

Performance and Tuning: Query Tuning

This book contains the following parts and chapters:

- Part 1, Basic Concepts
  - 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.

- Part 2, Tuning Query Performance
  - Chapter 5, "Understanding the Query Optimizer," explains the process of query optimization, and how statistics are applied to search arguments and joins for queries.
  - Chapter 6, "Access Methods and Query Costing for Single Tables," describes how Adaptive Server accesses tables in queries that only involve a single table, and how the costs are estimated for various access methods.
  - Chapter 7, "Access Methods and Query Costing for Joins and Subqueries,"describes how Adaptive Server accesses tables during joins and subqueries and how the costs are determined.
  - Chapter 8, "Cursors and Performance," describes performance issues with cursors.
  - Chapter 9, "Indexing for Performance," provides guidelines and examples for choosing indexes.
  - Chapter 10, "Managing Statistics to Improve Performance," describes how to use the **update statistics** command to create and update statistics.
- Part 3, Parallel Query Concepts and Tuning
  - Chapter 11, "Introduction to Parallel Query Processing," introduces the concepts and resources required for parallel query processing.
  - Chapter 12, "Parallel Query Optimization," provides an in-depth look at the optimization of parallel queries.
  - Chapter 13, "Parallel Sorting," describes the use of parallel sorting for queries and for creating indexes.
- Part 4, Query Tuning Tools
  - Chapter 14, "Introduction to Query Tuning Tools," presents an overview of query tuning tools and describes how these tools can interact.
  - Chapter 15, "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 16, "Using the set statistics Commands," explains the commands that provide information about query execution.
  - Chapter 17, "Using set showplan," provides examples of **showplan** messages.

- Chapter 18, "Tuning with dbcc traceon," explains how to use the **dbcc traceon** commands to analyze query optimization problems.
- Chapter 19, "Statistics Tables and Displaying Statistics with optdiag," describes the tables that store statistics and the output of the **optdiag** command that displays the statistics used by the query optimizer.
- Chapter 20, "Advanced Optimizing Tools," describes advanced tools for tuning query performance.

• Part 5, Query Tuning With Abstract Plans

- Chapter 21, "Introduction to Abstract Plans," provides an overview of abstract plans and how they can be used to solve query optimization problems.
- Chapter 22, "Creating and Using Abstract Plans," describes the commands that can be used to save and use abstract plans.
- Chapter 23, "Managing Abstract Plans with System Procedures," describes the system procedures that manage abstract plans and abstract plan groups.
- Chapter 24, "Abstract Plan Language Reference," describes the abstract plan language.
- Chapter 25, "A User's Guide to Abstract Query Plans," provides an introduction to writing abstract plans for specific types of queries and to using abstract plans to detect changes in query optimization due to configuration or system changes.

Performance and Tuning: System Tuning

• Part 6, Locking

- Chapter 26, "Locking in Adaptive Server," describes the types of locks that Adaptive Server uses and what types of locks are acquired during query processing.
- Chapter 27, "Locking Commands," describes the commands that set locking schemes for tables and control isolation levels and other locking behavior during query processing.
- Chapter 28, "Reporting on Locks," describes the system procedures that report on locks and lock contention.
- Chapter 29, "Locking Configuration and Tuning," describes the impact of locking on performance and describes the tools to analyze locking problems and configure locking.

- Part 7, Application Maintenance
  - Chapter 30, "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 31, "Setting Space Management Properties," describes how space management properties can be set for tables to improve performance and reduce the frequency of maintenance operations on tables and indexes.
- System-Level Tuning
  - Chapter 32, "Memory Use and Performance," describes how Adaptive Server uses memory for the procedure and data caches.
  - Chapter 33, "Controlling Physical Data Placement," describes the uses of segments and partitions for controlling the physical placement of data on storage devices.
  - Chapter 34, "Tuning Asynchronous Prefetch," describes how asynchronous prefetch improves performance for queries that perform large amounts of disk I/O.
  - Chapter 35, "tempdb Performance Issues," stresses the importance of the temporary database, *tempdb*, and provides suggestions for improving its performance.
  - Chapter 37, "How Adaptive Server Uses Engines and CPUs," describes how client processes are scheduled on engines in Adaptive Server.
  - Chapter 38, "Distributing Engine Resources Between Tasks," describes how to assign execution precedence to specific applications.
  - Chapter 39, "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 the Sybase Technical Library.

- The Adaptive Server installation documentation for your platform – describes installation, upgrade, and configuration procedures for all Adaptive Server and related Sybase products.
- *What's New in Adaptive Server Enterprise?* – Describes the new features in Adaptive Server version 12, the system changes added to support those features, and the changes that may affect your existing applications.
- *Transact-SQL User's Guide* – Documents Transact-SQL, Sybase's enhanced version of the relational database language. This manual serves as a textbook for beginning users of the database management system. This manual also contains descriptions of the *pubs2* and *pubs3* sample databases.
- *System Administration Guide* – Provides in-depth information about administering servers and databases. This manual includes instructions and guidelines for managing physical resources, security, user and system databases, and specifying character conversion, international language, and sort order settings.
- *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.
- *Error Messages* and *Troubleshooting Guide* – Explains how to resolve frequently occurring error messages and describes solutions to system problems frequently encountered by users.
- *Component Integration Services User's Guide* – Explains how to use the Adaptive Server Component Integration Services feature to connect remote Sybase and non-Sybase databases.

- *Java in Adaptive Server Enterprise* – Describes how to install and use Java classes as datatypes and user-defined functions in the Adaptive Server database.
- *Using Sybase Failover in a High Availability System* – Provides instructions for using Sybase's Failover to configure an Adaptive Server as a companion server in a high availability system.
- *Using Adaptive Server Distributed Transaction Management Features* – Explains how to configure, use, and troubleshoot Adaptive Server DTM Features in distributed transaction processing environments.
- *XA Interface Integration Guide for CICS, Encina, and TUXEDO* Provides instructions for using Sybase's DTM XA Interface with X/Open XA transaction managers.
- *Adaptive Server Glossary* – Defines technical terms used in the Adaptive Server documentation.

Other Sources of Information

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

- Technical Library CD contains product manuals and technical documents and is included with your software. The DynaText browser (included on the Technical Library CD) allows you to access technical information about your product in an easy-to-use format.

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

- Technical Library Web site includes the Product Manuals site, which is an HTML version of the Technical Library CD that you can access using a standard Web browser. In addition, you'll find links to the Technical Documents Web site (formerly known as Tech Info Library), the Solved Cases page, and Sybase/Powersoft newsgroups.

  To access the Technical Library Web site, go to support.sybase.com, click the Electronic Support Services tab, and select a link under the Technical Library heading.

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 *italics*.	*master* database
Variables, or words that stand for values that you fill in, are in *italics*.	`select` *column_name* `from` *table_name* `where` *search_conditions*
Parentheses are to be typed as part of the command.	`compute` *row_aggregate* (*column_name*)
Curly braces indicate that you must choose at least one of the enclosed options. Do not type the braces.	`{cash, check, credit}`
Brackets mean choosing one or more of the enclosed options is optional. Do not type the brackets.	`[anchovies]`
The vertical bar means you may select only one of the options shown.	`{die_on_your_feet | live_on_your_knees | live_on_your_feet}`
The comma means you may choose as many of the options shown as you like, separating your choices with commas to be typed as part of the command.	`[extra_cheese, avocados, sour_cream]`

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

Element	Example

An ellipsis (...) means that you can **repeat** the last unit as many times as you like.	`buy thing = price [cash | check | credit]`
	` [, thing = price [cash | check | credit]]...`
	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.

- 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 19, "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
expression	Can include constants, literals, functions, column identifiers, variables, or parameters
logical expression	An expression that returns TRUE, FALSE, or UNKNOWN
constant expression	An expression that always returns the same value, such as “5+3” or “ABCDE”
float_expr	Any floating-point expression or expression that implicitly converts to a floating value
integer_expr	Any integer expression, or an expression that implicitly converts to an integer value
numeric_expr	Any numeric expression that returns a single value
char_expr	Any expression that returns a single character-type value
binary_expression	An expression that returns a single *binary* or *varbinary* 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.

Locking

26 Locking in Adaptive Server

This chapter discusses basic locking concepts and the locking schemes and types of locks used in Adaptive Server.

This chapter contains the following sections:

The following chapters provide more information on locking:

- Chapter 27, "Locking Commands,"describes commands that affect locking: specifying the locking scheme for tables, choosing an isolation level for a session or query, the **lock table** command, and server or session level lock timeouts periods.
- Chapter 28, "Reporting on Locks," describes commands for reporting on locks and locking behavior, including **sp_who**, **sp_lock**, and **sp_object_stats.**
- Chapter 29, "Locking Configuration and Tuning," describes performance considerations and suggestions and configuration parameters that affect locking.

How Locking Affects Performance

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

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

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

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

Overview of Locking

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

For example, in Figure 26-1, transactions T1 and T2 are attempting to access data at approximately the same time. T1 is updating values in a column, while T2 needs to report the sum of the values.

T1	Event Sequence	T2
`begin transaction`	T1 and T2 start.	`begin transaction`

`update account`
`set balance = balance - 100`
`where acct_number = 25`
	T1 updates *balance* for one account by subtracting $100.	

	T2 queries the sum balance, which is off by $100 at this point in time—should it return results now, or wait until T1 ends?	`select sum(balance)`
		`from account`
		`where acct_number < 50`
		
		`commit transaction`

`update account`
`set balance = balance + 100`
`where acct_number = 45`
	T1 updates balance of the other account by adding the $100.	

`commit transaction`	T1 ends.	

Figure 26-1: Consistency levels in transactions

If transaction T2 runs before T1 starts or after T1 completes, either execution of T2 returns the correct 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 correct 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 it 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.

Granularity of Locks and Locking Schemes

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 column of data. Such extremes affect the concurrency (number of users that can access the data) and locking overhead (amount of work to process lock requests) in the server. Adaptive Server supports locking at the table, page, and row level.

By locking at higher levels of granularity, the amount of work required to obtain and manage locks is reduced. If a query needs to read or update many rows in a table:

- It can acquire just one table-level lock
- It can acquire a lock for each page that contained one of the required rows
- It can acquire a lock on each row

Less overall work is required to use a table-level lock, but large-scale locks can degrade performance, by making other users wait until locks are released. Decreasing the lock size makes more of the data accessible to other users. However, finer granularity 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 provides these locking schemes:

- Allpages locking, which locks datapages and index pages
- Datapages locking, which locks only the data pages
- Datarows locking, which locks only the data rows

For each locking scheme, Adaptive Server can choose to lock the entire table for queries that acquire many page or row locks, or can lock only the affected pages or rows.

Allpages Locking

Allpages locking locks both data pages and index pages. When a query updates a value in a row in an allpages-locked table, the data page is locked with an exclusive lock. Any index pages affected by the update are also locked with exclusive locks. These locks are transactional, meaning that they are held until the end of the transaction.

Figure 26-2 shows the locks acquired on data pages and indexes while a new row is being inserted into an allpages-locked table.

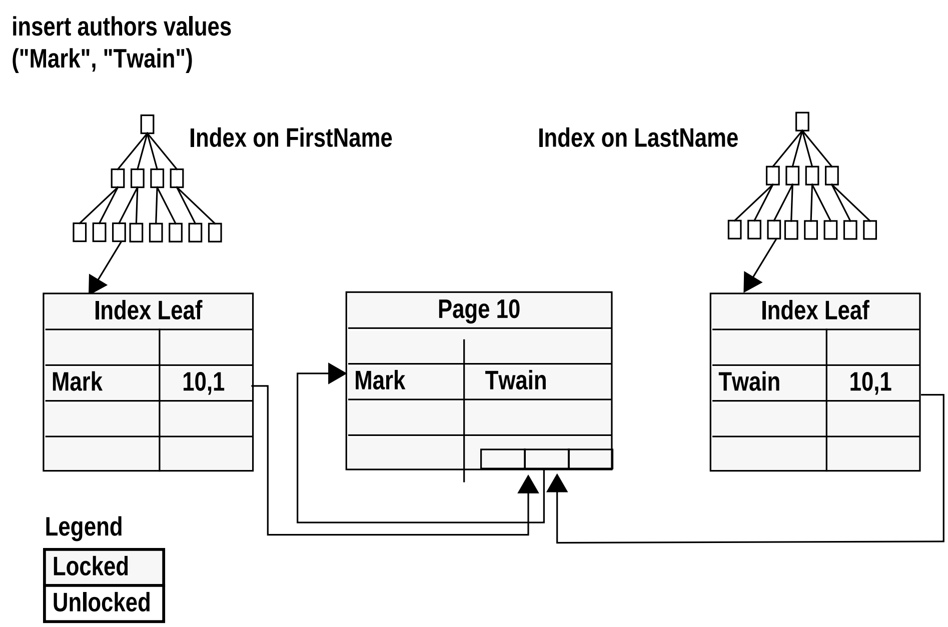


Figure 26-2: Locks held during allpages locking

In many cases, the concurrency problems that result from allpages locking arise from the index page locks, rather than the locks on the data pages themselves. Data pages have longer rows than indexes, and often have a small number of rows per page. If index keys are short, an index page can store between 100 and 200 keys. An exclusive lock on an index page can block other users who need to access any of the rows referenced by the index page, a far greater number of rows than on a locked data page.

Datapages Locking

In datapages locking, entire data pages are still locked, but index pages are not locked. When a row needs to be changed on a data page, that page is locked, and the lock is held until the end of the transaction. The updates to the index pages are performed using latches, which are nontransactional. Latches are held only as long as required to perform the physical changes to the page and are then released immediately. Index page entries are implicitly locked by locking the data page. No transactional locks are held on index pages. For more information on latches, see "Latches" on page 26-14.

Figure 26-3 shows an insert into a datapages-locked table. Only the affected data page is locked.

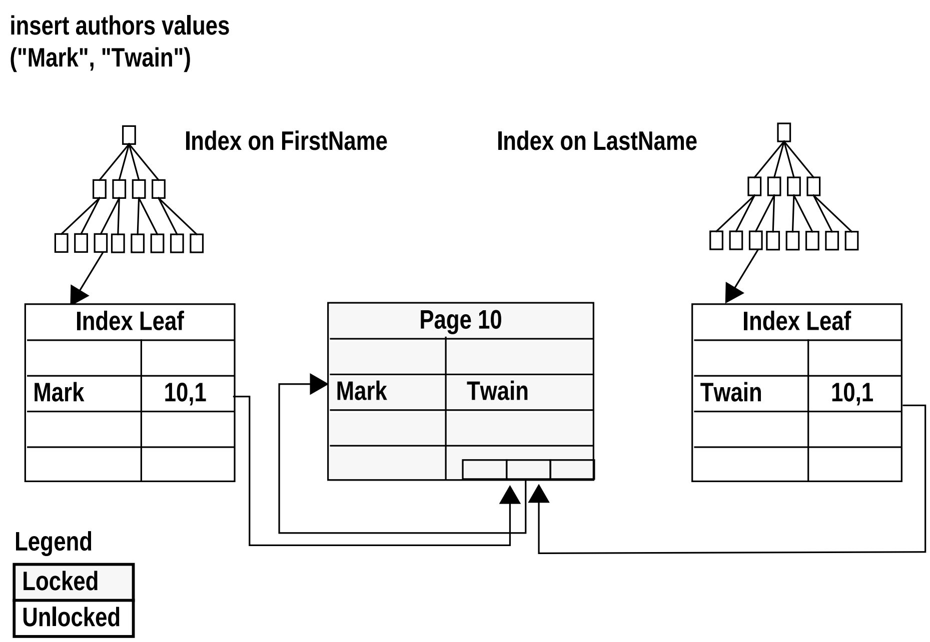


Figure 26-3: Locks held during datapages locking

Datarows Locking

In datarows locking, row-level locks are acquired on individual rows on data pages. Index rows and pages are not locked. When a row needs to be changed on a data page, a nontransactional latch is acquired on the page. The latch is held while the physical change is made to the data page, and then the latch is released. The lock on the data row is held until the end of the transaction. The index rows are updated, using latches on the index page, but are not locked. Index entries are implicitly locked by acquiring a lock on the data row.

Figure 26-4 shows an insert into a datarows-locked table. Only the affected data row is locked.

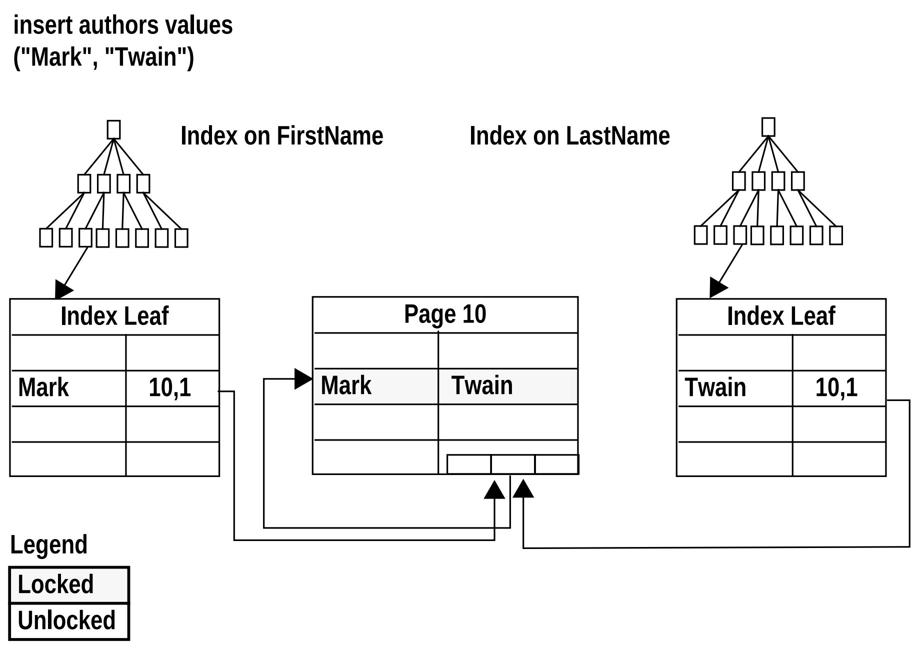


Figure 26-4: Locks held during datarows locking

Types of Locks in Adaptive Server

Adaptive Server has two levels of locking:

- For tables that use allpages locking or datapages locking, either page locks or table locks.
- For tables that use datarows locking, either row locks or table locks

Page or row locks are less restrictive (or smaller) than table locks. A page lock locks all the rows on data page or an index page; a table lock locks an entire table. A row lock locks only a single row on a page. Adaptive Server uses page or row locks whenever possible to reduce contention and to improve concurrency.

Adaptive Server uses a table lock to provide more efficient locking when an entire table or a large number of pages or rows 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 performs a table scan and acquires a table lock. For example, the following statement acquires a table lock:

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

If an **update** or **delete** statement uses an index, it begins by acquiring page or row locks. It tries to acquire a table lock only when a large number of pages or rows are affected. 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 or row locks than allowed by the lock promotion threshold, Adaptive Server tries to issue a table lock. If it succeeds, the page or row locks are no longer necessary and are released. See "Configuring Locks and Lock Promotion Thresholds" on page 29-5 for more information.

Adaptive Server 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 27-6 for more information. Applications can explicitly request a table lock with the **lock table** command.

Page and Row Locks

The following describes the types of page and row locks:

- **Shared locks**

  Adaptive Server applies **shared lock**s for read operations. If a shared lock has been applied to a data page or data row or to 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 or row until all shared locks on the page or row are released. This means that many transactions can simultaneously read the page or row, but no transaction can change data on the page or row while a shared lock exists. Transactions that need an exclusive lock wait or "block" for the release of the shared locks before continuing.

By default, Adaptive Server releases shared locks after it finishes scanning the page or row. It does not hold shared locks until the statement is completed or until the end of the transaction unless requested to do so by the user. For more details on how shared locks are applied, see "Locking for select Queries at Isolation Level 1" on page 26-26.

- **Exclusive locks**

  Adaptive Server applies an **exclusive lock** for a data modification operation. When a transaction gets an exclusive lock, other transactions cannot acquire a lock of any kind on the page or row 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 phase of an **update**, **delete**, or **fetch** (for cursors declared **for update**) operation while the page or row is being read. The update lock allows shared locks on the page or row, but does not allow other update or exclusive locks. Update locks help avoid deadlocks and lock contention. If the page or row needs to be changed, the update lock is promoted to an exclusive lock as soon as no other shared locks exist on the page or row.

In general, read operations acquire shared locks, and write operations acquire exclusive locks. For operations that delete or update data, Adaptive Server applies page-level or row-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 examples in Table 26-1 show what kind of page or row locks Adaptive Server uses for basic SQL statements. For these examples, there is an index *acct_number*, but no index on *balance*.

Table 26-1: Page locks and row locks

Statement	Allpages-Locked Table	Datarows-Locked Table
`select balance from account where acct_number = 25`	Shared page lock	Shared row lock
`insert account values (34, 500)`	Exclusive page lock on data page and exclusive page lock on leaf-level index page	Exclusive row lock
`delete account where acct_number = 25`	Update page locks followed by exclusive page locks on data pages and exclusive page locks on leaf-level index pages	Update row locks followed by exclusive row locks on each affected row
`update account set balance = 0 where acct_number = 25`	Update page lock on data page and exclusive page lock on data page	Update row lock followed by exclusive row lock

Table Locks

The following describes the types of table locks.

- **Intent lock**

  An **intent lock** indicates that page-level or row-level locks are currently held on a table. Adaptive Server applies an intent table lock with each shared or exclusive page or row lock, so an intent lock can be either an exclusive lock or a 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 page or row locks are in effect for the transaction.

- **Shared lock**

  This lock is similar to a shared page or 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. A **create nonclustered index** command also acquires a shared table lock.

- **Exclusive lock**

  This lock is similar to an exclusive page or row lock, except it affects the entire table. For example, Adaptive Server applies an exclusive table lock during a **create clustered index** command. **update** and **delete** statements require exclusive table locks if their search arguments do not reference indexed columns of the object.

The examples in Table 26-2 show the respective page, row, and table locks of page or row locks Adaptive Server uses for basic SQL statements. For these examples, there is an index *acct_num*.

Table 26-2: Table locks applied during query processing

Statement	Allpages-Locked Table	Datarows-Locked Table
`select balance from account where acct_number = 25`	Intent shared table lock Shared page lock	Intent shared table lock Shared row lock
`insert account values (34, 500)`	Intent exclusive table lock Exclusive page lock on data page Exclusive page lock on leaf index pages	Intent exclusive table lock Exclusive row lock
`delete account where acct_number = 25`	Intent exclusive table lock Update page locks followed by exclusive page locks on data pages and leaf-level index pages	Intent exclusive table lock Update row locks followed by exclusive row locks on data rows
`update account set balance = 0 where acct_number = 25`	With an index on *acct_number*, intent exclusive table lock Update page locks followed by exclusive page locks on data pages and leaf-level index pages With no index, exclusive table lock	With an index on *acct_number*, intent exclusive table lock Update row locks followed by exclusive row locks on data rows With no index, exclusive table lock

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, page, or row. Since many readers can all hold shared locks on a given page, row, or table, tasks that require exclusive locks are queued after a task that already holds a shared lock. Adaptive Server allows up to three readers' tasks to skip over a queued update task.

After a write transaction has been skipped over by three tasks or families (in the case of queries running in parallel) that acquire 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 26-5.

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 Locking with Serial Execution

Figure 26-5 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 26-5:

- 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 three 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.

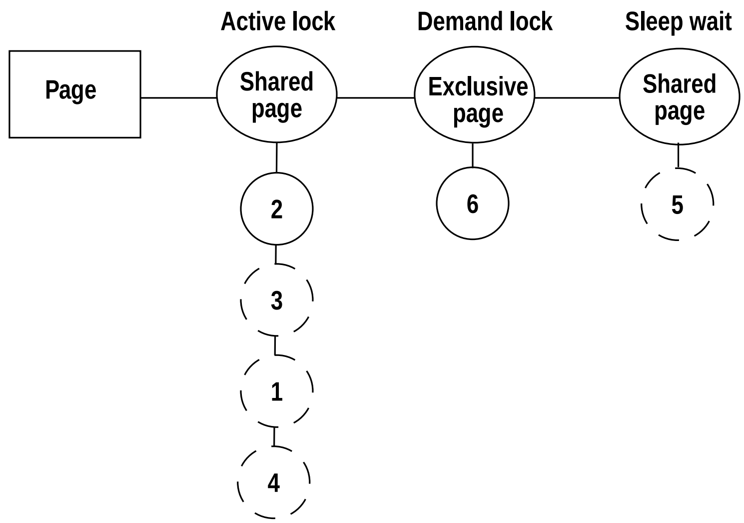


Figure 26-5: Demand locking with serial query execution

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 reads from three families (or a total of three serial tasks and families combined) before granting the exclusive lock.

Figure 26-6 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 26-6:

- Originally, worker process 1:3 (worker process 3 from a family with family ID 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.

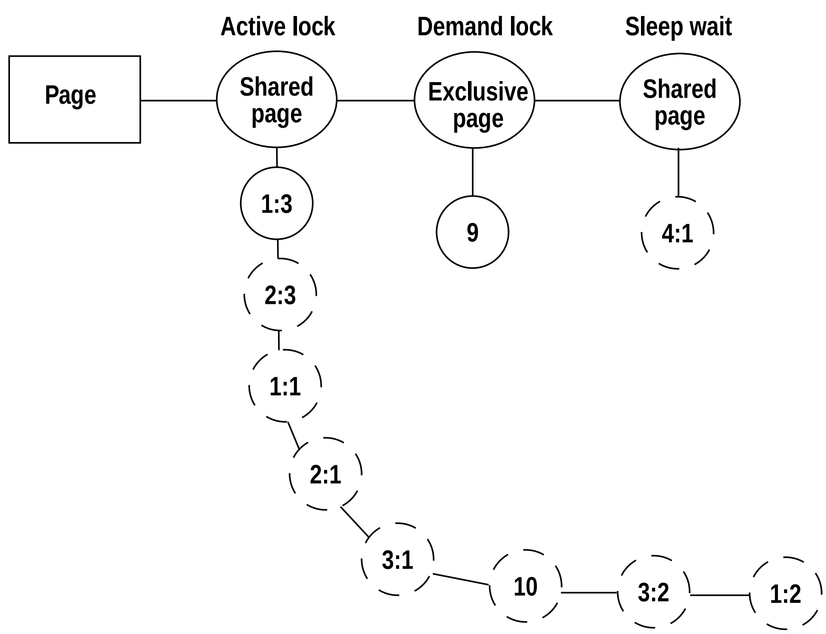


Figure 26-6: Demand locking with parallel query execution

Range Locking for Serializable Reads

Rows that can appear or disappear from a results set are called phantoms. Some queries that require **phantom** protection (queries at isolation level 3) use range locks.

Isolation level 3 requires serializable reads within the transaction. A query at isolation level 3 that performs two read operations with the same query clauses should return the same set of results each time. No other task can be allowed to:

- Modify one of the result rows so that it no longer qualifies for the serializable read transaction, by updating or deleting the row

- Modify a row that is not included in the serializable read result set so that the row now qualifies, or insert a row that would qualify for the result set

Adaptive Server uses range locks, infinity key locks, and next-key locks to protect against phantoms on data-only-locked tables. Allpages-locked tables protect against phantoms by holding locks on the index pages for the serializable read transaction.

When a query at isolation level 3 (serializable read) performs a range scan using an index, all the keys that satisfy the query clause are locked for the duration of the transaction. Also, the key that immediately follows the range is locked, to prevent new values from being added at the end of the range. If there is no next value in the table, an **infinity key lock** is used as the next key, to ensure that no rows are added after the last key in the table.

Range locks can be shared, update, or exclusive locks; depending on the locking scheme, they are either row locks or page locks. **sp_lock** output shows "Fam dur, Range" in the *context* column for range locks. For infinity key locks, **sp_lock** shows a lock on a nonexistent row, row 0 of the root index page and "Fam dur, Inf key" in the *context* column.

Every transaction that performs an insert or update to a data-only-locked table checks for range locks.

Latches

Latches are nontransactional synchronization mechanisms used to guarantee the physical consistency of a page. While rows are being inserted, updated or deleted, only one Adaptive Server process can have access to the page at the same time. Latches are used for datapages and datarows locking but not for allpages locking.

The most important distinction between a lock and a latch is the duration:

- A lock can persist for a long period of time: while a page is being scanned, while a disk read or network write takes place, for the duration of a statement, or for the duration of a transaction.
- A latch is held only for the time required to insert or move a few bytes on a data page, to copy pointers, columns or rows, or to acquire a latch on another index page.

Lock Compatibility and Lock Sufficiency

Two basic concepts underlie issues of locking and concurrency:

- Lock compatibility: if task holds a lock on a page or row, can another row also hold a lock on the page or row?
- Lock sufficiency: for the current task, is the current lock held on a page or row sufficient if the task needs to access the page again?

Lock compatibility affects performance when users needs to acquire a lock on a row or page, and that row or page is already locked by another user with an incompatible lock. The task that needs the lock waits, or blocks, until the incompatible locks are released.

Lock sufficiency works with lock compatibility. If a lock is sufficient, the task does not need to acquire a different type of lock. For example, if a task updates a row in a transaction, it holds an exclusive lock. If the task then selects from the row before committing the transaction, the exclusive lock on the row is sufficient; the task does not need to make an additional lock request. The opposite case is not true: if a task holds a shared lock on a page or row, and wants to update the row, the task may need to wait to acquire its exclusive lock if other tasks also hold shared locks on the page.

Table 26-3 summarizes the information about lock compatibility, showing when locks can be acquired immediately.

Table 26-3: Lock compatibility

	Can another process immediately acquire:				
If one process has:	**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

Table 26-4 shows the lock sufficiency matrix.

Table 26-4: Lock sufficiency

If a task has:	Is that lock sufficient if the task needs: A Shared Lock	An Update Lock	An Exclusive Lock
A Shared Lock	Yes	No	No
An Update Lock	Yes	Yes	No
An Exclusive Lock	Yes	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 interactions that are not permitted while concurrent transactions are executing—that is, whether transactions are isolated from each other, or if they can read or update information in use by another transaction. Higher isolation levels include the restrictions imposed by the lower levels.

The isolation levels are shown in Table 26-5, and described in more detail on the following pages.

Table 26-5: Transaction isolation levels

Number	Name	Description
0	**read uncommitted**	The transaction is allowed to read uncommitted changes to data.
1	**read committed**	The transaction is allowed to read only committed changes to data.
2	**repeatable read**	The transaction can repeat the same query, and no rows that have been read by the transaction will have been updated or deleted.
3	**serializable read**	The transaction can repeat the same query, and receive exactly the same results. No rows can be inserted that would appear in the result set.

You can choose the isolation level for all **select** queries during a session, or you can choose the isolation level for a specific query or table in a transaction.

At all isolation levels, all updates acquire exclusive locks and hold them for the duration of the transaction.

➤ ***Note***

For tables that use the allpages locking scheme, requesting isolation level 2 also enforces isolation level 3.

Isolation Level 0, Read Uncommitted

Level 0, also known as **read uncommitted**, allows a task to read uncommitted changes to data in the database. This is also known as a dirty read, since the task can display results that are later rolled back. Figure 26-7 shows a select query performing 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 26-7: 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.The **update** statement in transaction T3 acquires an exclusive lock on *account*. However, 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, because T4 does not hold any locks on the pages it reads.

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

- Allowing another task to read rows, pages, or tables that have exclusive locks; that is, to read uncommitted changes to data.
- Not applying shared locks on rows, pages or tables being searched.

Any data modifications that are performed by T4 while the isolation level is set to 0 acquire exclusive locks at the row, page, or table level, and block if the data they need to change is locked.

If the table uses allpages locking, 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.

Applications that can use dirty reads may see better concurrency and reduced deadlocks than when the same data is accessed at a higher isolation level. If transaction T4 requires only an estimate 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 make 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.

sp_sysmon reports on these factors. See "Dirty Read Behavior" on page 39-79.

Isolation Level 1, Read Committed

Level 1, also known as **read committed**, prevents dirty reads. Queries at level 1 can read only committed changes to data. At isolation level 1, if a transaction needs to read a row that has been modified by an incomplete transaction in another session, the transaction waits until the first transaction completes (either commits or rolls back.)

For example, compare Figure 26-8, showing a transaction executed at isolation level 1, to Figure 26-7, showing a dirty read transaction.

T5	Event Sequence	T6
`begin transaction`	T5 and T6 start.	`begin transaction`

`update account`
`set balance = balance - 100`
`where acct_number = 25`
	T5 updates account after getting exclusive lock.	

	T6 tries to get shared lock to query account but must wait until T5 releases its lock.	`select sum(balance)`
		`from account`
		`where acct_number < 50`

`rollback transaction`	T5 ends and releases its exclusive lock.	
	T6 gets shared lock, queries account, and ends.	`commit transaction`

Figure 26-8: Transaction isolation level 1 prevents dirty reads

When the **update** statement in transaction T5 executes, Adaptive Server applies an exclusive lock (a row-level or page-level lock if *acct_number* is indexed; otherwise, a table-level lock) on *account*.

If T5 holds an exclusive table lock, T6 blocks trying to acquire its shared intent table lock. If T5 holds exclusive page or exclusive row locks, T6 can begin executing, but is blocked when it tries to acquire a shared lock on a page or row locked by T5. 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, Repeatable Read

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. Isolation level 2 is supported only on data-only-locked tables. In a session at isolation level 2, isolation level 3 is also enforced on any tables that use the allpages locking scheme. Figure 26-9 shows a nonrepeatable read in a transaction at isolation level 1.

T7	Event Sequence	T8
`begin transaction`	T7 and T8 start.	`begin transaction`

`select balance`
`from account`
`where acct_number = 25`
	T7 queries the balance for one account.	

	T8 updates the balance for that same account.	`update account`
		`set balance = balance - 100`
		`where acct_number = 25`

	T8 ends.	`commit transaction`

`select balance`
`from account`
`where acct_number = 25`
	T7 makes same query as before and gets different results.	

`commit transaction`	T7 ends.	

Figure 26-9: 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 would produce different results. Isolation level 2 blocks transaction T8 from executing. It would also block a transaction that attempted to delete the selected row.

Isolation Level 3, Serializable Reads

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. In Figure 26-10, transaction T9, operating at isolation level 1, sees a phantom row in the second query.

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

Figure 26-10: 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 prevents phantoms by:

- Applying exclusive locks on rows, pages, or tables being changed. It holds those locks until the end of the transaction.
- Applying shared locks on rows, pages, or tables being searched. It holds those locks until the end of the transaction.
- Using range locks or infinity key locks for certain queries on data-only-locked tables.

Holding the 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 26-10, with the same transaction executed at isolation level 3, as shown in Figure 26-11.

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

Figure 26-11: 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 the locks until the end of T11. The **insert** in T12 cannot get its exclusive lock until T11 releases its shared locks. If T11 is a long transaction, T12 (and other transactions) may wait for longer periods of time. 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 row, page or table.

Using 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.

Lock Types and Duration During Query Processing

The types and the duration of locks acquired during query processing depend on the type of command, the locking scheme of the table, and the isolation level at which the command is run.

The lock duration depends on the isolation level and the type of query. Lock duration can be one of the following:

- Scan duration – Locks are released when the scan moves off the row or page, for row or page locks, or when the scan of the table completes, for table locks.
- Statement duration – Locks are released when the statement execution completes.
- Transaction duration – Locks are released when the transaction completes.

Table 26-6 shows the types of locks acquired by queries at different isolation levels, for each locking scheme for queries that do not use cursors. Table 26-7 shows information for cursor-based queries.

Table 26-6: Lock type and duration without cursors

Statement	Isolation Level	Locking Scheme	Table Lock	Data Page Lock	Index Page Lock	Data Row Lock	Duration
select **readtext** any type of scan	0	allpages datapages datarows	- - -	- - -	- - -	- - -	No locks are acquired.
	1 2 with **noholdlock** 3 with **noholdlock**	allpages datapages datarows	IS IS IS	S * -	S - -	- - *	* Depends on setting of **read committed with lock**. See "Locking for select Queries at Isolation Level 1" on page 26-26.
	2	allpages datapages datarows	IS IS IS	S S -	S - -	- - S	Locks are released at the end of the transaction. See "Isolation Level 2 and Allpages-Locked Tables" on page 26-27.
select via index scan	3 1 with **holdlock** 2 with **holdlock**	allpages datapages datarows	IS IS IS	S S -	S - -	- - S	Locks are released at the end of the transaction.
select via table scan	3 1 with **holdlock** 2 with **holdlock**	allpages datapages datarows	IS S S	S - -	- - -	- - -	Locks are released at the end of the transaction.
insert	0, 1, 2, 3	allpages datapages datarows	IX IX IX	X X -	X - -	- - X	Locks are released at the end of the transaction.
writetext	0, 1, 2, 3	allpages datapages datarows	IX IX IX	X X -	- - -	- - X	Locks are held on first text page or row; locks released at the end of the transaction.
delete **update** any type of scan	0, 1, 2	allpages datapages datarows	IX IX IX	U, X U, X -	U, X - -	- - U, X	"U" locks are released after the statement completes. "IX" and "X" locks are released at the end of the transaction.
delete **update** via index scan	3	allpages datapages datarows	IX IX IX	U, X U, X -	U, X - -	- - U, X	"U" locks are released after the statement completes. "IX" and "X" locks are released at the end of the transaction.
delete **update** via table scan	3	allpages datapages datarows	IX X X	U, X - -	- - -	- - -	Locks are released at the end of the transaction.
Key: IS intent shared, IX intent exclusive, S shared, U update, X exclusive							

Table 26-7: Lock type and duration with cursors

Statement	Isolation Level	Locking Scheme	Table Lock	Data Page Lock	Index Page Lock	Data Row Lock	Duration
select (without **for** clause) **select... for read only**	0	allpages datapages datarows	- - -	- - -	- - -	- - -	No locks are acquired.
	1 2 with **noholdlock** 3 with **noholdlock**	allpages datapages datarows	IS IS IS	S * -	S - -	- - *	* Depends on setting of **read committed with lock**. See “Locking for select Queries at Isolation Level 1” on page 26-26.
	2, 3 1 with **holdlock** 2 with **holdlock**	allpages datapages datarows	IS IS IS	S S -	S - -	- - S	Locks become transactional after the cursor moves out of the page / row. Locks are released at the end of the transaction.
select...for update	1	allpages datapages datarows	IX IX IX	U, X U, X -	X - -	- - U, X	“U” locks are released after the cursor moves out of the page / row. “IX” and “X” locks are released at the end of the transaction.
select...for update with shared	1	allpages datapages datarows	IX IX IX	S, X S, X -	X - -	- - S, X	“S” locks are released after the cursor moves out of page / row. “IX” and “X” locks are released at the end of the transaction.
select...for update	2, 3, 1 **holdlock** 2, **holdlock**	allpages datapages datarows	IX IX IX	U, X U, X -	X - -	- - U, X	Locks become transactional after the cursor moves out of the page / row. Locks are released at the end of the transaction.
select...for update with shared	2, 3 1 with **holdlock** 2 with **holdlock**	allpages datapages datarows	IX IX IX	S, X S, X -	X - -	- - S, X	Locks become transactional after the cursor moves out of the page / row. Locks are released at the end of the transaction.

Key: IS intent shared, IX intent exclusive, S shared, U update, X exclusive

Lock Types During *create index* Commands

Table 26-8 describes the types of locks applied by Adaptive Server for **create index** statements:

Table 26-8: Summary of locks for insert and create index statements

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

Locking for *select* Queries at Isolation Level 1

When a **select** query on an allpages-locked table performs a table scan at isolation level 1, it first acquires a shared intent lock on the table and then acquires a shared lock on the first data page. It locks the next data page, and drops the lock on the first page, so that the locks "walk through" the result set. As soon as the query completes, the lock on the last data page is released, and then the table-level lock is released. Similarly, during index scans on an allpages-locked table, overlapping locks are held as the scan descends from the index root page to the data page. Locks are also held on the outer table of a join while matching rows from inner table are scanned.

select queries on data-only-locked tables first acquire a shared intent table lock. Locking behavior on the data pages and data rows is configurable with the parameter **read committed with lock**, as follows:

- If **read committed with lock** is set to 0 (the default) then **select** queries read the column values with instant-duration page or row locks. The required column values or pointers for the row are read into memory, and the lock is released. Locks are not held on the outer tables of joins while rows from the inner tables are accessed. This reduces deadlocking and improves concurrency.

  If a **select** query needs to read a row that is locked with an incompatible lock, the query still blocks on that row until the incompatible lock is released. Setting **read committed with lock** to 0 does not affect the isolation level; only committed rows are returned to the user.

- If **read committed with lock** is set to 1, **select** queries acquire shared page locks on datapages-locked tables and shared row locks on datarows-locked tables. The lock on the first page or row is held, then the lock is acquired on the second page or row and the lock on the first page or row is dropped.

Cursors must be declared as read-only to avoid holding locks during scans when **read committed with lock** is set to 0. Any implicitly or explicitly updatable cursor on a data-only-locked table holds locks on the current page or row until the cursor moves off the row or page. When **read committed with lock** is set to 1, read-only cursors hold a shared page or row lock on the row at the cursor position.

read committed with lock does not affect locking behavior on allpages-locked tables. For information on setting the configuration parameter, see "read committed with lock" on page 17-77 in the *System Administration Guide*.

Table Scans and Isolation Levels 2 and 3

This section describes special considerations for locking during table scans at isolation levels 2 and 3.

Table Scans and Table Locks at Isolation Level 3

When a query performs a table scan at isolation level 3 on a data-only-locked table, a shared or exclusive table lock provides phantom protection and reduces the locking overhead of maintaining a large number of row or page locks. On an allpages-locked table, an isolation level 3 scan first acquires a shared or exclusive intent table lock and then acquires and holds page-level locks until the transaction completes or until the lock promotion threshold is reached and a table lock can be granted.

Isolation Level 2 and Allpages-Locked Tables

On allpages-locked tables, Adaptive Server supports isolation level 2 (repeatable reads) by also enforcing isolation level 3 (serializable reads). If transaction level 2 is set in a session, and an allpages-locked table is included in a query, isolation level 3 will also be applied on the allpages-locked tables. Transaction level 2 will be used on all data-only-locked tables in the session.

When Update Locks Are Not Required

All **update** and **delete** commands on an allpages-locked table first acquire an update lock on the data page and then change to an exclusive lock if the row meets the qualifications in the query.

Updates and delete commands on data-only-locked tables do not first acquire update locks when:

- The query includes search arguments for every key in the index chosen by the query, so that the index unambiguously qualifies the row, and
- The query does not contain an **or** clause.

Updates and deletes that meet these requirements immediately acquire an exclusive lock on the data page or data row. This reduces lock overhead.

Locking During *or* Processing

In some cases, queries using **or** clauses are processed as a union of more than one query. Although some rows may match more than one of the **or** conditions, each row must be returned only once. Different indexes can be used for each **or** clause. If any of the clauses do not have a useful index, the query is performed using a table scan.

The table's locking scheme and the isolation level affect how **or** processing is performed and the types and duration of locks that are held during the query.

Processing *or* Queries for Allpages-Locked Tables

If the **or** query uses the OR Strategy (different **or** clauses might match the same rows), query processing retrieves the row IDs and matching key values from the index and stores them in a worktable, holding shared locks on the index pages containing the rows. When all row IDs have been retrieved, the worktable is sorted to remove duplicate values. Then, the worktable is scanned, and the row IDs are used to retrieve the data rows, acquiring shared locks on the data pages. The index and data page locks are released at the end of the statement (for isolation level 1) or at the end of the transaction (for isolation levels 2 and 3).

If the **or** query has no possibility of returning duplicate rows, no worktable sort is needed. At isolation level 1, locks on the data pages are released as soon as the scan moves off the page.

Processing *or* Queries for Data-Only-Locked Tables

On data-only-locked tables, the type and duration of locks acquired for **or** queries using the OR Strategy (when multiple clauses might match the same rows) depend on the isolation level.

Processing or Queries at Isolation Levels 1 and 2

No locks are acquired on the index pages or rows of data-only-locked tables while row IDs are being retrieved from indexes and copied to a worktable. After the worktable is sorted to remove duplicate values, the data rows are requalified when the row IDs are used to read data from the table. If any rows were deleted, they are not returned. If any rows were updated, they are requalified by applying the full set of query clauses to them. The locks are released when the row qualification completes, for isolation level 1, or at the end of the transaction, for isolation level 2.

Processing or Queries at Isolation Level 3

Isolation level 3 requires serializable reads. At this isolation level, **or** queries obtain locks on the data pages or data rows during the first phase of **or** processing, as the worktable is being populated. These locks are held until the transaction completes. Requalification of rows is not required.

Skipping Uncommitted Inserts During Selects

select queries on data-only-locked tables do not block on uncommitted inserts when the following conditions are true:

- The table uses datarows locking, and
- The isolation level is 1 or 2.

Pseudo Column-Level Locking

During concurrent transactions that involve **select** queries and **update** commands, pseudo column-level locking can allow some queries to return values from locked rows, and can allow other queries to avoid blocking on locked rows that do not qualify. Pseudo column-level locking can reduce blocking:

- When the **select** query does not reference columns on which there is an uncommitted update.
- When the **where** clause of a **select** query references one or more columns affected by an uncommitted update, but the row does not qualify due to conditions in other clauses.
- When neither the old nor new value of the updated column qualifies, and an index containing the updated column is being used.

Select Queries That Do Not Reference the Updated Column

A **select** query on a datarows-locked table can return values without blocking, even though a row is exclusively locked when:

- The query does not reference an updated column in the select list or any clauses (**where**, **having**, **group by**, **order by** or **compute**), and
- The query does not use an index that includes the updated column

Transaction T14 in Figure 26-12 requests information about a row that is locked by T13. However, since T14 does not include the updated column in the result set or as a search argument, T14 does not block on T13’s exclusive row lock.

T13	Event Sequence	T14
`begin transaction`	T13 and T14 start.	`begin transaction`

`update accounts`
`set balance = 50`
`where acct_number = 35`
	T13 updates accounts and holds an exclusive row lock.	

	T14 queries the same row in accounts, but does not access the updated column. T14 does not block.	`select lname, fname, phone`
		`from accounts`
		`where acct_number = 35`
		`commit transaction`

`commit transaction`		

Figure 26-12: Pseudo-column-level locking with mutually-exclusive columns

If T14 uses an index that includes the updated column (for example, *acct_number, balance*), the query blocks trying to read the index row.

For **select** queries to avoid blocking when they do not reference updated columns, all of the following conditions must be met:

- The table must use datarows locking.
- The columns referenced in the **select** query must be among the first 32 columns of the table.
- The **select** query must run at isolation level 1.
- The **select** query must not use an index that contains the updated column.
- The configuration parameter **read committed with lock** must be set to 0, the default value.

Using Alternative Predicates to Skip Nonqualifying Rows

When a **select** query includes multiple **where** clauses linked with **and**, Adaptive Server can apply the qualification for any columns that have not been affected by an uncommitted update of a row. If the row does not qualify because of one of the clauses on an unmodified column, the row does not need to be returned, so the query does not block.

If the row qualifies when the conditions on the unmodified columns have been checked, and the conditions described in the next section, "Qualifying Old and New Values for Uncommitted Updates" does not allow the query to proceed, then the query blocks until the lock is released.

For example, transaction T15 in Figure 26-13 updates *balance*, while transaction T16 includes *balance* in the result set and in a search clause. However, T15 does not update the *branch* column, so T16 can apply that search argument.

Since the *branch* value in the row affected by T15 is not 77, the row does not qualify, and the row is skipped, as shown. If T15 updated a row where *branch* equals 77, a **select** query would block until T15 either commits or rolls back.

T15	Event Sequence	T16
`begin transaction`	T15 and T16 start.	`begin transaction`

`update accounts`
`set balance = 80`
`where acct_number = 20`
`and branch = 23`
	T15 updates accounts and holds an exclusive row lock.	

	T16 queries accounts, but does not block because the branch qualification can be applied.	`select acct_number, balance`
		`from accounts`
		`where balance < 50`
		`and branch = 77`
		`commit tran`

`commit transaction`		

Figure 26-13: Pseudo-column-level locking with multiple predicates

For **select** queries to avoid blocking when they reference columns in addition to columns that are being updated, all of the following conditions must be met:

- The table must use datarows or datapages locking.
- At least one of the search clauses of the **select** query must be on a column that among the first 32 columns of the table.
- The **select** query must run at isolation level 1 or 2.
- The configuration parameter **read committed with lock** must be set to 0, the default value.

Qualifying Old and New Values for Uncommitted Updates

If a **select** query includes conditions on a column affected by an uncommitted update, and the query uses an index on the updated column, the query can examine both the old and new values for the column:

- If neither the old or new value meets the search criteria, the row can be skipped, and the query does not block.
- If either the old or new value, or both of them qualify, the query blocks In Figure 26-14, if the original balance is $80, and the new balance is $90, the row can be skipped, as shown. If either of the values is less than $50, T18 must wait until T17 completes.

T17	Event Sequence	T18
`begin transaction`	T17 and T18 start.	`begin transaction`

`update accounts`
`set balance = balance + 10`
`where acct_number = 20`
	T17 updates accounts and holds an exclusive row lock; the original balance was 80, so the new balance is 90.	

	T18 queries accounts using an index that includes balance. It does not block since balance does not qualify.	`select acct_number, balance`
		`from accounts`
		`where balance < 50`
		`commit tran`

`commit transaction`		

Figure 26-14: Checking old and new values for an uncommitted update

For **select** queries to avoid blocking when old and new values of uncommitted updates do not qualify, all of the following conditions must be met:

- The table must use datarows or datapages locking.
- At least one of the search clauses of the **select** query must be on a column that among the first 32 columns of the table.
- The **select** query must run at isolation level 1 or 2.
- The index used for the **select** query must include the updated column.

- The configuration parameter **read committed with lock** must be set to 0, the default value.

Suggestions to Reduce Contention

To help reduce lock contention between **update** and **select** queries:

- Use datarows or datapages locking for tables with lock contention due to updates and selects.
- If tables have more than 32 columns, make the first 32 columns the columns that are most frequently used as search arguments and in other query clauses.
- Select only needed columns. Avoid using **select *** when all columns are not needed by the application.
- Use any available predicates for **select** queries. When a table uses datapages locking, the information about updated columns is kept for the entire page, so that if a transaction updates some columns in one row, and other columns in another row on the same page, any select query that needs to access that page must avoid using any of the updated columns.

27 Locking Commands

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

This chapter contains the following sections:

Specifying the Locking Scheme for a Table

The locking schemes in Adaptive Server provide you with the flexibility to choose the best locking scheme for each table in your application and to adapt the locking scheme for a table if contention or performance requires a change. The tools for specifying locking schemes are:

- **sp_configure**, to specify a server-wide default locking scheme
- The **create table** command, to specify the locking scheme for newly created tables
- The **alter table** command, to change the locking scheme for a table to any other locking scheme
- The **select into** command, to specify the locking scheme for a table created by selecting results from other tables

Specifying a Server-Wide Locking Scheme with *sp_configure*

The **lock scheme** configuration parameter sets the locking scheme to be used for any new table, if the **create table** command does not specify the lock scheme.

To see the current locking scheme, use:

```
sp_configure "lock scheme"
```

```
Parameter Name   Default     Memory Used Config Value Run Value
---------------- ----------- ----------- ------------ -----------
lock scheme      allpages              0 datarows    datarows
```

The syntax for changing the locking scheme is:

```
sp_configure "lock scheme", 0,
      {allpages | datapages | datarows }
```

This command sets the default lock scheme for the server to datapages:

```
sp_configure "lock scheme", 0, datapages
```

When you first install Adaptive Server, **lock scheme** is set to **allpages**.

Specifying a Locking Scheme with *create table*

You can specify the locking scheme for a new table with the **create table** command. The syntax is:

```
create table table_name (column_name_list)
   [ lock {datarows | datapages | allpages } ]
```

If you do not specify the lock scheme for a table, the default value for your server is used, as determined by the setting of the **lock scheme** configuration parameter.

This command specifies datarows locking for the *new_publishers* table:

```
create table new_publishers
(pub_id     char(4)     not null,
 pub_name   varchar(40) null,
 city       varchar(20) null,
 state      char(2)     null)
lock datarows
```

Specifying the locking scheme with **create table** overrides the default server-wide setting. See "Specifying a Server-Wide Locking Scheme with sp_configure" on page 27-1 for information on setting the server-wide locking scheme.

Changing a Locking Scheme with *alter table*

Use the **alter table** command to change the locking scheme for a table. The syntax is:

```
alter table table_name
   lock { allpages | datapages | datarows }
```

This command changes the locking scheme for the *titles* table to datarows locking:

```
alter table titles lock datarows
```

alter table supports changing from one locking scheme to any other locking scheme. Changing from allpages locking to data-only locking requires copying the data rows to new pages and re-creating any indexes on the table. The operation takes several steps and requires sufficient space to make the copy of the table and indexes. The time required depends on the size of the table and the number of indexes.

Changing from datapages locking to datarows locking or vice versa does not require copying data pages and rebuilding indexes. Switching between data-only locking schemes only updates system tables, and completes in a few seconds.

➤ ***Note***

You cannot use data-only locking for tables that have rows that are at, or near, the maximum length of 1962 (including the two bytes for the offset table). For data-only-locked tables with only fixed-length columns, the maximum user data row size is 1960 bytes (including the 2 bytes for the offset table). Tables with variable-length columns require 2 additional bytes for each column that is variable-length (this includes columns that allow nulls.) See Chapter 15, “Determining or Estimating the Sizes of Tables and Indexes,” for information on rows and row overhead.

Before and After Changing Locking Schemes

Before you change from allpages locking to data-only locking or vice versa, the following steps are recommended:

- If the table is partitioned, and **update statistics** has not been run since major data modifications to the table, run **update statistics** on the table that you plan to alter. **alter table...lock** performs better with accurate statistics for partitioned tables. Note that changing the locking scheme does not affect the distribution of data on partitions; rows in partition 1 are copied to partition 1 in the copy of the table.
- Perform a database dump.
- Set any space management properties that should be applied to the copy of the table or its rebuilt indexes. See Chapter 31, “Setting Space Management Properties,” for more information.
- Determine if there is enough space. See “Determining the Space Available for Maintenance Activities” on page 30-10.

- If any of the tables in the database are partitioned and require a parallel sort:
  - Use **sp_dboption** to set the database option **select into/bulkcopy/pllsort** to true and run **checkpoint** in the database.
  - Set your configuration for optimum parallel sort performance. See Chapter 13, “Parallel Sorting.”

After *alter table* Completes

- Run **dbcc checktable** on the table and **dbcc checkalloc** on the database to insure database consistency.
- Perform a database dump.

➤ ***Note***

After changing the locking scheme from allpages locking to data-only locking or vice versa, the use of the **dump transaction** command to back up the transaction log is prohibited. You must first perform a full database dump.

Expense of Switching to or from Allpages Locking

Switching from allpages locking to data-only locking or vice versa is an expensive operation, in terms of I/O cost. The amount of time required depends on the size of the table and the number of indexes that must be re-created. Most of the cost comes from the I/O required to copy the tables and re-create the indexes. Some logging is also required.

The **alter table...lock** command performs the following actions when moving from allpages locking to data-only locking or from data-only locking to allpages locking:

- Copies all rows in the table to new data pages, formatting rows according to the new format. If you are changing to data-only locking, any data rows of less than 10 bytes are padded to 10 bytes during this step. If you are changing to allpages locking from data-only locking, extra padding is stripped from rows of less than 10 bytes.
- Drops and re-creates all indexes on the table.

- Deletes the old set of table pages.
- Updates the system tables to indicate the new locking scheme.
- Updates a counter maintained for the table, to cause the recompilation of query plans.

If a clustered index exists on the table, rows are copied in clustered index key order onto the new data pages. If no clustered index exists, the rows are copied in page-chain order for an allpages-locking to data-only-locking conversion.

The entire **alter table...lock** command is performed as a single transaction to ensure recoverability. An exclusive table lock is held on the table for the duration of the transaction.

Switching from datapages locking to datarows locking or vice versa does not require copying pages or re-creating indexes. It only updates the system tables. Setting **sp_dboption "select into/bulkcopy/pllsort"** is not required.

Sort Performance During *alter table*

If the table being altered is partitioned, parallel sorting can be used while rebuilding the indexes. **alter table** performance can be greatly improved if the data cache and server are configured for optimal parallel sort performance.

During **alter table**, the indexes are re-created one at a time. If your system has enough engines, data cache, and I/O throughput to handle simultaneous **create index** operations, you can reduce the overall time required to change locking schemes by doing the following:

- Drop the nonclustered indexes
- Alter the locking scheme
- Configure for best parallel sort performance
- Re-create two or more nonclustered indexes at once

For information on configuring for parallel sort, see Chapter 13, “Parallel Sorting.”

Specifying a Locking Scheme with *select into*

You can specify a locking scheme when you create a new table, using the **select into** command. The syntax is:

```
select [all | distinct] select_list
       into [[database.]owner.]table_name
       lock {datarows | datapages | allpages }
from ...
```

If you do not specify a locking scheme with **select into**, the new table uses the server-wide default locking scheme, as defined by the configuration parameter **lock scheme**.

This command specifies datarows locking for the table it creates:

```
select title_id, title, price
into bus_titles
lock datarows
from titles
where type = "business"
```

Temporary tables created with the *#tablename* form of naming are single-user tables, so lock contention is not an issue. For temporary tables that can be shared among multiple users, that is, tables created with *tempdb..tablename*, any locking scheme can be used.

Controlling Isolation Levels

You can set the transaction isolation level used by **select** commands:

- For all queries in the session, with the **set transaction isolation level** command
- For an individual query, with the **at isolation** clause
- For specific tables in a query, with the **holdlock**, **noholdlock**, and **shared** keywords

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.

➤ ***Note***

If you use transaction isolation level 2 (repeatable reads) on allpages-locked tables, isolation level 3 (serializable reads) is also enforced.

For more information on isolation levels, see “How Isolation Levels Affect Locking” on page 26-16.

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 Adaptive Server default isolation level of 1, or if you have used the **set transaction isolation level** command to specify level 0 or 2, 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*.

Syntax for Query-Level and Table-Level Locking Options

The **holdlock**, **noholdlock**, and **shared** options can be specified for each table in a **select** statement, with the **at isolation** clause applied 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 | 0 ] |
       [ read committed | 1 ] |
       [ repeatable read | 2 ]|
       [ serializable | 3 ] ]
```

Here is the syntax for the **readtext** command:

```
readtext [[database.]owner.]table_name.column_name
   text_pointer offset size
   [holdlock | noholdlock] [readpast]
   [using {bytes | chars | characters}]
   [at isolation {
       [ read uncommitted | 0 ] |
       [ read committed | 1 ] |
       [ repeatable read | 2 ]|
       [ serializable | 3 ] } ]
```

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

You can override a session's locking level by applying the **holdlock**, **noholdlock**, and **shared** options to individual tables in **select** or **readtext** commands:

Level to Use	Keyword	Effect
1	**noholdlock**	Do not hold locks until the end of the transaction; use from level 3 to enforce level 1
2, 3	**holdlock**	Hold shared locks until the transaction completes; use from level 1 to enforce level 3
N/A	**shared**	Applies shared rather than update locks for **select** statements 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** in a query while isolation level 0 is in effect for the session, Adaptive Server issues a warning and ignores the **holdlock** clause, not acquiring locks as the query executes. If you specify **holdlock** and **read uncommitted**, Adaptive Server prints an error message, and the query is not executed.

Using the *at isolation* Clause

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

Level to Use	Option	Effect
0	**read uncommitted**	Reads uncommitted changes; use from level 1, 2, or 3 queries to perform dirty reads (level 0).
1	**read committed**	Reads only committed changes; wait for locks to be released; use from level 0 to read only committed changes, but without holding locks.
2	**repeatable read**	Holds shared locks until the transaction completes; use from level 0 or level 1 queries to enforce level 2.
3	**serializable**	Holds shared locks until the transaction completes; use from level 1 or level 2 queries to enforce level 3.

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 some queries require higher levels of isolation, you can selectively enforce the higher isolation level using clauses in the **select** statement:

- Use **repeatable read** to enforce level 2
- Use **holdlock** or **at isolation serializable** to enforce level 3

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** 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.

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 2 or 3. If any queries in those transactions do not need to hold shared locks until the end of the transaction, you can specify **noholdlock** with those queries to improve concurrency.

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 releases the locks when the scan moves off the page or row:

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

Using *read uncommitted*

If your session isolation level is 1, 2, 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 27-13 for more information.

Readpast Locking

Readpast locking allows select and readtext queries to silently skip all rows or pages locked with incompatible locks. The queries do not block, terminate, or return error or advisory messages to the user. It is largely designed to be used in queue-processing applications. In general, these applications allow queries to return the first unlocked row that meets query qualifications. An example might be an application tracking calls for service: the query needs to find the row with the earliest timestamp that is not locked by another repair representative. For more information on readpast locking, see Chapter 19, "Locking Commands and Options," in the *Transact-SQL User's Guide*.

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 2 or 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 may 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 **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 the **holdlock** keyword in conjunction with **shared** after each table or view name. **holdlock** must precede **shared** in the **select** statement. For example:

```
declare authors_crsr 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 when defining an updatable cursor:

- If you do not specify either option, the cursor holds an update lock on the row or on the page containing the current row. Other users cannot update, through a cursor or otherwise, the row at the cursor position (for datarows-locked tables) or any row on this page (for allpages and datapages-locked tables). Other users can declare a cursor on the same tables you use for your cursor, and can read data, but they cannot get an update or exclusive lock on your current row or page.
- If you specify the **shared** option, the cursor holds a shared lock on the current row or on the page containing the currently fetched row. Other users cannot update, through a cursor or otherwise, the current row, or the rows on this page. They can, however, read the row or rows on the page.
- If you specify the **holdlock** option, you hold update locks on all the rows or 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 rows or 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 rows or pages.

- If you specify both options, the cursor holds shared locks on all the rows or pages fetched (if not using transactions) or on the rows or pages fetched since the last commit or rollback. Other users cannot update, through a cursor or otherwise, currently fetched rows or pages.

Additional Locking Commands

lock table Command

In transactions, you can explicitly lock a table with the **lock table** command.

- To immediately lock the entire table, rather than waiting for lock promotion to take effect.
- When the query or transactions uses multiple scans, and none of the scans locks a sufficient number of pages or rows to trigger lock promotion, but the total number of locks is very large.
- When large tables, especially those using datarows locking, need to be accessed at transaction level 2 or 3, and lock promotion is likely to be blocked by other tasks. Using lock table can prevent running out of locks.

The table locks are released at the end of the transaction.

lock table allows you to specify a wait period. If the table lock cannot be granted within the wait period, an error message is printed, but the transaction is not rolled back. See **lock table** in the *Adaptive Server Reference Manual* for an example of a stored procedure that uses lock timeouts, and checks for an error message. The procedure continues to execute if it was run by the System Administrator, and returns an error message to other users.

Lock Timeouts

You can specify the time that a task waits for a lock:

- At the server level, with the **lock wait period** configuration parameter
- For a session or in a stored procedure, with the **set lock wait** command
- For a **lock table** command

See Chapter 19, “Locking Commands and Options,” in the *Transact-SQL User’s Guide* for more information on these commands.

Except for lock table, a task that attempts to acquire a lock and fails to acquire it within the time period returns an error message and the transaction is rolled back. Using lock timeouts can be useful for removing tasks that acquire some locks, and then wait for long periods of time blocking other users. However, since transactions are rolled back, and users may simply resubmit their queries, timing out a transaction means that the work needs to be repeated.

You can use **sp_sysmon** to monitor the number of tasks that exceed the time limit while waiting for a lock. See “Lock Timeout Information” on page 39-70.

28 Reporting on Locks

This chapter discusses the tools that report on locks and locking behavior. It contains the following sections:

Using *sp_who*, *sp_lock*, and *sp_familylock*

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*

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_spid* or *block_xloid* column shows the process ID of the task or transaction 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.

➤ ***Note***

The output for **sp_lock** in this chapter omits the *class* column to increase readability. The *class* column reports either the names of cursors that hold locks or "Non Cursor Lock."

Viewing Locks with *sp_lock*

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

```
sp_lock
```

```
fid spid loid locktype         table_id   page  row dbname   context
--- ---- ---- ---------------- ---------- ----- --- -------- -------------------
  0  15   30  Ex_intent         208003772     0   0 sales    Fam dur
  0  15   30  Ex_page           208003772  2400   0 sales    Fam dur, Ind pg
  0  15   30  Ex_page           208003772  2404   0 sales    Fam dur, Ind pg
  0  15   30  Ex_page-blk       208003772   946   0 sales    Fam dur
  0  30   60  Ex_intent         208003772     0   0 sales    Fam dur
  0  30   60  Ex_page           208003772   997   0 sales    Fam dur
  0  30   60  Ex_page           208003772  2405   0 sales    Fam dur, Ind pg
  0  30   60  Ex_page           208003772  2406   0 sales    Fam dur, Ind pg
  0  35   70  Sh_intent          16003088     0   0 sales    Fam dur
  0  35   70  Sh_page            16003088  1096   0 sales    Fam dur, Inf key
  0  35   70  Sh_page            16003088  3102   0 sales    Fam dur, Range
  0  35   70  Sh_page            16003088  3113   0 sales    Fam dur, Range
  0  35   70  Sh_page            16003088  3365   0 sales    Fam dur, Range
  0  35   70  Sh_page            16003088  3604   0 sales    Fam dur, Range
  0  49   98  Sh_intent         464004684     0   0 master   Fam dur
  0  50  100  Ex_intent         176003658     0   0 stock    Fam dur
  0  50  100  Ex_row            176003658 36773   8 stock    Fam dur
  0  50  100  Ex_intent         208003772     0   0 stock    Fam dur
  0  50  100  Ex_row            208003772 70483   1 stock    Fam dur
  0  50  100  Ex_row            208003772 70483   2 stock    Fam dur
  0  50  100  Ex_row            208003772 70483   3 stock    Fam dur
  0  50  100  Ex_row            208003772 70483   5 stock    Fam dur
  0  50  100  Ex_row            208003772 70483   8 stock    Fam dur
  0  50  100  Ex_row            208003772 70483   9 stock    Fam dur
 32  13   64  Sh_page           240003886 17264   0 stock
 32  16   64  Sh_page           240003886  4376   0 stock
 32  17   64  Sh_page           240003886  7207   0 stock
 32  18   64  Sh_page           240003886 12766   0 stock
 32  18   64  Sh_page           240003886 12767   0 stock
 32  18   64  Sh_page           240003886 12808   0 stock
 32  19   64  Sh_page           240003886 22367   0 stock
 32  32   64  Sh_intent          16003088     0   0 stock    Fam dur
 32  32   64  Sh_intent          48003202     0   0 stock    Fam dur
 32  32   64  Sh_intent          80003316     0   0 stock    Fam dur
 32  32   64  Sh_intent         112003430     0   0 stock    Fam dur
 32  32   64  Sh_intent         176003658     0   0 stock    Fam dur
 32  32   64  Sh_intent         208003772     0   0 stock    Fam dur
 32  32   64  Sh_intent         240003886     0   0 stock    Fam dur
 51  23  102  Sh_page           208003772   945   0 sales
 51  51  102  Sh_intent          16003088     0   0 sales    Fam dur
 51  51  102  Sh_intent          48003202     0   0 sales    Fam dur
 51  51  102  Sh_intent         176003658     0   0 sales    Fam dur
 51  51  102  Sh_intent         208003772     0   0 sales    Fam dur
```

This example shows the lock status of serial processes and two parallel processes:

- *spid* 15 hold an exclusive intent lock on a table, one data page lock, and two index page locks. The lock on the data page shows a lock type of “Ex_page-blk” which means this task is blocking another task.
- *spid* 30 holds an exclusive intent lock on a table, one lock on a data page, and two locks on index pages.

- *spid* 35 is performing a range query at isolation level 3. It holds range locks on several pages and an infinity key lock.
- *spid* 49 is the task that ran **sp_lock**; it holds a shared intent lock on the *spt_values* table in *master* while it runs.
- *spid* 50 holds intent locks on two tables, and several row locks.
- *fid* 32 shows several *spid*s holding locks: the parent process (*spid* 32) holds shared intent locks on 7 tables, while the worker processes hold shared page locks on one of the tables.
- *fid* 51 has two *spids* holding locks: the parent process (*spid* 51) holds shared intent locks on the 4 tables in the query. *spid* 23 holds a shared page lock.

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" or "row."

A "blk" suffix indicates that this process is blocking another process that needs to acquire a lock; *spid* 15 is blocking another process. 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. See "Demand Locks" on page 26-10 for more information on demand locks.

The *context* column consists of one or more of the following values:

- "Fam dur" means that the task will hold the lock until the query completes, that is, for the duration of the family of worker processes. Shared intent locks are an example of Fam dur locks. For a parallel query, the coordinating process always acquires 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 family duration locks on all of the changed data pages. Worker processes can hold family duration locks when the query operates at isolation level 3.
- "Ind pg" indicates locks on index pages (allpages-locked tables only).
- "Inf key" indicates an infinity key lock, used on data-only-locked tables for some range queries at transaction isolation level 3.
- "Range" indicates a range lock, used for some range queries at transaction isolation level 3.

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

`sp_lock 30`

```
fid spid loid locktype         table_id   page  row dbname   context
--- ---- ---- ---------------- ---------- ----- --- -------- ------------------
  0  30   60  Ex_intent         208003772     0   0 sales    Fam dur
  0  30   60  Ex_page           208003772   997   0 sales    Fam dur
  0  30   60  Ex_page           208003772  2405   0 sales    Fam dur, Ind pg
  0  30   60  Ex_page           208003772  2406   0 sales    Fam dur, Ind pg
```

If the *spid* you specify is also the *fid* for a family of processes, **sp_who** prints information for all of the processes.

You can also request information about locks on two *spids*:

`sp_lock 30, 15`

```
fid spid loid locktype         table_id   page  row dbname   context
--- ---- ---- ---------------- ---------- ----- --- -------- ------------------
  0  15   30  Ex_intent         208003772     0   0 sales    Fam dur
  0  15   30  Ex_page           208003772  2400   0 sales    Fam dur, Ind pg
  0  15   30  Ex_page           208003772  2404   0 sales    Fam dur, Ind pg
  0  15   30  Ex_page-blk       208003772   946   0 sales    Fam dur
  0  30   60  Ex_intent         208003772     0   0 sales    Fam dur
  0  30   60  Ex_page           208003772   997   0 sales    Fam dur
  0  30   60  Ex_page           208003772  2405   0 sales    Fam dur, Ind pg
  0  30   60  Ex_page           208003772  2406   0 sales    Fam dur, Ind pg
```

Viewing Locks with *sp_familylock*

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 51`

```
fid spid loid locktype         table_id   page  row dbname   context
--- ---- ---- ---------------- ---------- ----- --- -------- ------------------
 51  23  102  Sh_page           208003772   945   0 sales
 51  51  102  Sh_intent          16003088     0   0 sales    Fam dur
 51  51  102  Sh_intent          48003202     0   0 sales    Fam dur
 51  51  102  Sh_intent         176003658     0   0 sales    Fam dur
 51  51  102  Sh_intent         208003772     0   0 sales    Fam dur
```

You can also specify two IDs for **sp_familylock.**

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.

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 same page or table locked by the other process. 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.

Server-Side vs. Application-Side Deadlocks

When tasks deadlock in Adaptive Server, a deadlock detection mechanism finds the deadlock, rolls back one of the transactions, and sends messages to the user and to the Adaptive Server error log. It is possible to induce application-side deadlock situations in which a client opens multiple connections, and these client connections wait for locks held by the other connection of the same application. These are not true server-side deadlocks and cannot be detected by Adaptive Server deadlock detection mechanisms.

Application Deadlock Example

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 can 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 the logic 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.

Server Task Deadlocks

Figure 28-1 shows an example of a deadlock between two processes.

T199	Event Sequence	T200
begin transaction	T19 and T20 start.	begin transaction
update savings set balance = balance - 250 where acct_number = 25	T19 gets exclusive lock on savings while T20 gets exclusive lock on checking.	update checking set balance = balance - 75 where acct_number = 45
update checking set balance = balance + 250 where acct_number = 45	T19 waits for T 20 to release its lock while T20 waits for T19 to release its lock;	update savings set balance = balance + 75 where acct_number = 25
commit transaction	deadlock occurs.	commit transaction

Figure 28-1: Deadlocks in transactions

Table 28-2 shows the deadlock graphically. See Figure 28-3 on page 28-9 for an illustration of a deadlock involving parallel processes.

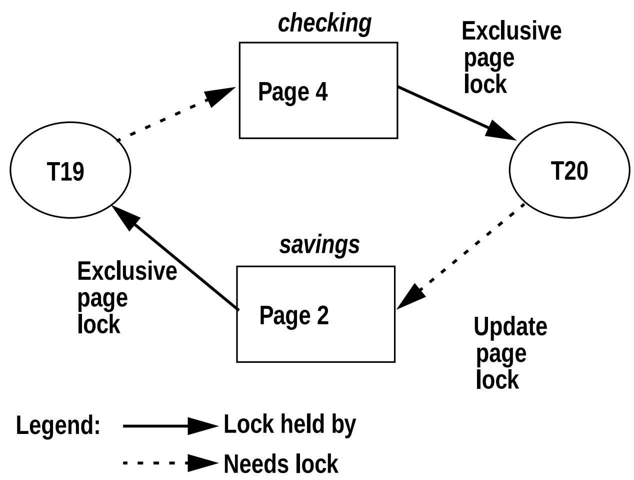


Figure 28-2: A deadlock between two processes

If transactions T19 and T20 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 process to move forward. The example above shows two data modification statements that deadlock; deadlocks can also occur between a process holding and needing shared locks, and one holding and needing exclusive locks.

In a multiuser situation, each 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.

Deadlocks and Parallel Queries

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 those 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.

In some cases, deadlocks between serial processes and families involve a level of indirection. For example, if a task holds an exclusive lock on *tableA* and needs a lock on *tableB*, but a worker process holds a family-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 28-3 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_tbl*, at transaction level 3.
- The serial task identified by *spid* 17 (T_{17}) is performing inserts to *stock_tbl* and *sales_tbl* 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*.
- T_{17} holds an exclusive lock on page 634 of *sales_tbl*. T_{17} needs an exclusive lock on page 10862, which it cannot acquire until $W_{8\ 9}$ releases its shared lock.
- The worker process $W_{8\ 10}$ needs a shared lock on page 634, which it cannot acquire until T_{17} releases its exclusive lock.

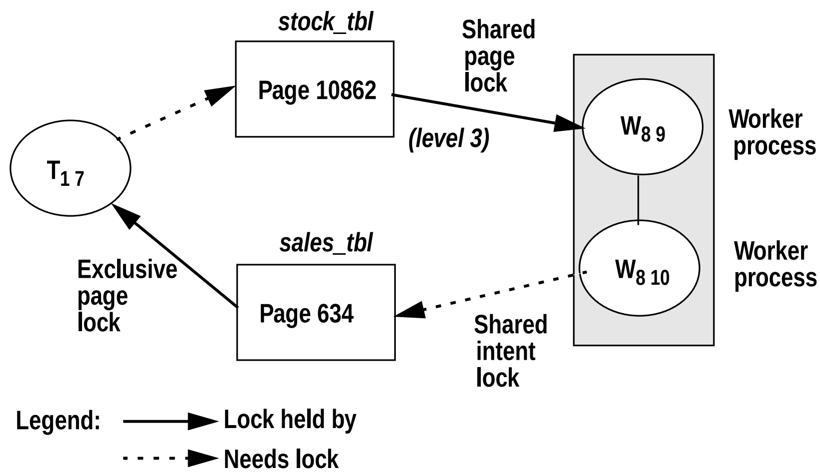


Figure 28-3: A deadlock involving a family of worker processes

Printing Deadlock Information to the Error Log

Server-side deadlocks are detected and reported to the application by Adaptive Server and in the server's error log. The error message sent to the application is error 1205. The message sent to the error log, by default, merely identifies that a deadlock occurred. The numbering in the message indicates the number of deadlocks since the last boot of the server.

```
03:00000:00029:1999/03/15 13:16:38.19 server  Deadlock Id 11 detected
```

In this output, *fid* 0, *spid* 29 started the deadlock detection check, so its *fid* and *spid* values are used as the second and third values in the deadlock message. (The first value, 03, is the engine number.)

To get more information about the tasks that deadlock, set the configuration parameter **print deadlock information** to 1. More detailed deadlock messages are sent to the log and to the terminal session where the server was started. 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.

The deadlock messages contain detailed information giving

- The family and server-process IDs of the tasks involved
- The commands and tables involved in deadlocks; if a stored procedure was involved, the procedure name is shown
- The type of locks each task held, and the type of lock each task was trying to acquire

- The server login IDs (*suid* values)

In the following report, *spid* 29 is deadlocked with a parallel task, *fid* 94, *spid* 38. The deadlock involves exclusive versus shared lock requests on the *authors* table. *spid* 29 is chosen as the deadlock victim:

```
Deadlock Id 11: detected. 1 deadlock chain(s) involved.

Deadlock Id 11: Process (Familyid 94, 38) (suid 62) was executing a
SELECT command at line 1.
Deadlock Id 11: Process (Familyid 29, 29) (suid 56) was executing a
INSERT command at line 1.
SQL Text: insert authors (au_id, au_fname, au_lname) values
('A999999816', 'Bill', 'Dewart')

Deadlock Id 11: Process (Familyid 0, Spid 29) was waiting for a
'exclusive page' lock on page 1155 of the 'authors' table in database
8 but process (Familyid 94, Spid 38) already held a 'shared page' lock
on it.
Deadlock Id 11: Process (Familyid 94, Spid 38) was waiting for a 'shared
page' lock on page 2336 of the 'authors' table in database 8 but process
(Familyid 29, Spid 29) already held a 'exclusive page' lock on it.
Deadlock Id 11: Process (Familyid 0, 29) was chosen as the victim. End
of deadlock information.
```

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 changing the locking scheme, avoiding table locks, and not holding shared locks, are described in Chapter 29, “Locking Configuration and Tuning.”

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 28-1 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.

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.

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. **sp_sysmon** can help you tune deadlock checking behavior. See "Deadlock Detection" on page 39-69.

Identifying Tables Where Concurrency Is a Problem

sp_object_stats prints table-level information about lock contention. You can use it to:

- Report on all tables that have the highest contention level

- Report contention on tables in a single database
- Report contention on individual tables.

The syntax is:

```
sp_object_stats interval [, top_n
   [, dbname [, objname [, rpt_option ]]]]
```

To measure lock contention on all tables in all databases, specify only the interval. This example monitors lock contention for 20 minutes, and reports statistics on the 10 tables with the highest levels of contention:

```
sp_object_stats "00:20:00"
```

Additional arguments to **sp_object_stats** are as follows:

- *top_n* – Allows you to specify the number of tables to be included in the report. To report on the top 20 high-contention tables, for example, use:

  ```
  sp_object_stats "00:20:00", 20
  ```

- *dbname* – Prints statistics for the specified database.
- *objname* – Measures contention for the specified table.
- *rpt_option* – Specifies the report type:
  - **rpt_locks** reports grants, waits, deadlocks, and wait times for the tables with the highest contention. **rpt_locks** is the default.
  - **rpt_objlist** reports only the names of the objects with the highest level of lock activity.

Here is sample output for *titles*, which uses datapages locking:

```
Object Name: pubtune..titles (dbid=7, objid=208003772,lockscheme=Datapages)

 Page Locks     SH_PAGE                UP_PAGE               EX_PAGE
 ----------    ----------             ----------            ----------
 Grants:            94488                   4052                  4828
 Waits:               532                    500                   776
 Deadlocks:             4                      0                    24
 Wait-time:     20603764 ms            14265708 ms            2831556 ms
 Contention:         0.56%                 10.98%                13.79%

*** Consider altering pubtune..titles to Datarows locking.
```

Table 28-1 shows the meaning of the values.

Table 28-1: sp_object_stats output

Output Row	Value
Grants	The number of times the lock was granted immediately.
Waits	The number of times the task needing a lock had to wait.
Deadlocks	The number of deadlocks that occurred.
Contention	The percentage of times that a task had to wait or encountered a deadlock.
Wait-times	The total number of milliseconds that all tasks spent waiting for a lock.

sp_object_stats recommends changing the locking scheme when total contention on a table is more than 15 percent, as follows:

- If the table uses allpages locking, it recommends changing to datapages locking.
- If the table uses datapages locking, it recommends changing to datarows locking.

Lock Management Reporting Using *sp_sysmon*

Output from **sp_sysmon** gives statistics on locking 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 39-62.

Use Adaptive Server Monitor to pinpoint locking problems.

29 Locking Configuration and Tuning

This chapter discusses the types of locks used in Adaptive Server and the commands that can affect locking. It contains the following sections:

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

  Any time 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

  A 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. Using datapages or datarows locking or redesigning the way transactions access the data can help reduce deadlock 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 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* and *sp_object_stats* While Reducing Lock Contention

Many of the following sections suggest changing configuration parameters to reduce lock contention. Use **sp_object_stats** or **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 "Identifying Tables Where Concurrency Is a Problem" on page 28-11 for information on using **sp_object_stats**. See "Lock Management" on page 39-62 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 impact Adaptive Server's throughput and response time. You need to consider locking during database design, and monitor locking during application design. Solutions include changing the locking scheme for tables with high contention, or redesigning the application or tables that have the highest lock contention. For example:

- Add indexes to reduce contention, especially for deletes and updates
- Keep transactions short to reduce the time that locks are held
- Check for "hot spots," especially for inserts on allpages-locked heap tables

Adding Indexes to Reduce Contention

An **update** or **delete** statement that has no useful index on its search arguments performs a table scan and holds an exclusive table lock for the entire scan time. If the data modification task also updates other tables, it can be blocked by select queries or other updates. It may be blocked and have to wait while holding large numbers of locks. It can block or deadlock with other tasks. Creating a useful index for the query allows the data modification statement to use page or row 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.

	With page-level locking	With row-level locking
begin tran		
select balance from account holdlock where acct_number = 25	*Intent shared table lock Shared page lock* If the user goes to lunch now, no one can update rows on the page that holds this row.	*Intent shared table lock Shared row lock* If the user goes to lunch now, no one can update this row.
update account set balance = balance + 50 where acct_number = 25	*Intent exclusive table lock Update page lock on data page followed by exclusive page lock on data page* If the user goes to lunch now, no one can read rows on the page that holds this row.	*Intent exclusive table lock Update row lock on data page followed by exclusive row lock on data page* If the user goes to lunch now, no one can read this row.
commit tran		

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 update account set balance = balance + 50 where acct_number = 25 go	*isql batch sent to Adaptive Server Locks held waiting for commit*
update account set balance = balance - 50 where acct_number = 45 commit tran go	*isql batch sent to Adaptive Server Locks released*

Keeping transactions short is especially crucial for data modifications that affect nonclustered index keys on allpages-locked tables. 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.

Avoiding "Hot Spots"

Hot spots occur when all updates take place on a certain page, as in an allpages-locked 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% 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 %
```

Possible solutions are:

- Change the lock scheme to datapages or datarows locking. Since these locking schemes do not have chained data pages, they can allocate additional pages when blocking occurs for inserts.
- Partition the 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 33-16 for information about partitioning tables.
- 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 overhead for maintaining the physical order of the table's rows.

Additional Locking Guidelines

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

- Use the lowest level of locking required by each application. Use isolation level 2 or 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 inserts, updates, or 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 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.

Configuring Locks and Lock Promotion Thresholds

A System Administrator can configure:

- The total number of locks available to processes on Adaptive Server.
- The size of the lock hashtable and the number of spinlocks that protect the page/row lock hashtable, table lock hashtable, and address lock hashtable.
- The server-wide lock timeout limit, and the lock timeout limit for distributed transactions.

- Lock promotion thresholds, 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 17-71 and “max engine freelocks” on page 17-72 of the *System Administration Guide* for information on these parameters.

Configuring Adaptive Server’s Lock Limit

By default, Adaptive Server is configured with 5000 locks. A System Administrator can change this limit using **sp_configure**. For example:

```
sp_configure "number of locks", 25000
```

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 locking scheme and 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 20 locks for each active concurrent connection, plus 20 locks for each worker process. Consider increasing the number of locks:

- If you change tables to use datarows locking
- If queries run at isolation level 2 or 3, or use **serializable** or **holdlock**
- If you enable parallel query processing, especially for isolation level 2 or 3 queries
- If you perform many multirow updates
- If you increase lock promotion thresholds

Estimating *number of locks* for Data-Only-Locked Tables

Changing to data-only locking may require more locks or may reduce the number of locks required:

- Tables using datapages locking require fewer locks than tables using allpages locking, since queries on datapages-locked tables do not acquire separate locks on index pages.
- Tables using datarows locking can require a large number of locks. Although no locks are acquired on index pages for datarows-locked tables, data modification commands that affect

many rows may hold more locks. Queries running at transaction isolation level 2 or 3 can acquire and hold very large numbers of row locks.

Insert Commands and Locks

An insert with allpages locking requires *N*+1 locks, where *N* is the number of indexes. The same insert on a data-only-locked table locks only the data page or data row.

select Queries and Locks

Scans at transaction isolation level 1, with **read committed with lock** set to hold locks (1), acquire overlapping locks that roll through the rows or pages, so they hold, at most, two data page locks at a time. However, transaction isolation level 2 and 3 scans, especially those using datarows locking, can acquire and hold very large numbers of locks, especially when running in parallel. Using datarows locking, and assuming no blocking during lock promotion, the maximum number of locks that might be required for a single table scan is:

```
row lock promotion HWM * parallel_degree
```

If lock contention from exclusive locks prevents scans from promoting to a table lock, the scans can acquire a very large number of locks. Instead of configuring the number of locks to meet the extremely high locking demands for queries at isolation level 2 or 3, consider changing applications that affect large numbers of rows to use the **lock table** command. This command acquires a table lock without attempting to acquire individual page locks. See “lock table Command” on page 27-14 for information on using **lock table**.

Data Modification Commands and Locks

For tables that use the datarows locking scheme, data modification commands can require many more locks than data modification on allpages or datapages-locked tables. For example, a transaction that performs a large number of inserts into a heap table may acquire only a few page locks for an allpages-locked table, but will require one lock for each inserted row in a datarows-locked table. Similarly, transactions that update or delete large numbers of rows may acquire many more locks with datarows locking.

Configuring the Lock Hashtable

The configuration parameter **lock hashtable size** configures the number of buckets in the lock hash table. The default size of the lock hash table is appropriate for most installations. However, if you have a large number of users and have had to increase the **number of locks** parameter to avoid running out of locks, you should check the average hash chain length with **sp_sysmon** at peak periods. If the average length of the hash chains exceeds 4 or 5, consider increased the value of **lock hashtable size** to the next power of 2 from its current setting.

The hash chain length may be high during large insert batches, such as bulk copy operations. This is expected behavior, and does not require resetting the lock hash table size.

Setting Lock Promotion Thresholds

The lock promotion thresholds set the number of page or row locks permitted by a task or worker process before Adaptive Server attempts to escalate to a table lock on the object. You can set lock promotion thresholds 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.

➤ ***Note***

Lock promotion is always two-tiered: from page locks to table locks or from row locks to table locks. Row locks are never promoted to page locks.

Lock Promotion and Scan Sessions

Lock promotion occurs on a per-scan session basis. A **scan session** is how which Adaptive Server tracks scans of tables 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 or row locks when an entire table might eventually be needed. At first, a task acquires page or row locks, then attempts to escalate to a table lock when a scan session acquires more page or row 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 or row locks for a single transaction can exceed the lock promotion threshold, as long as no single scan session acquires more than the lock promotion threshold number of locks. Locks may persist throughout a transaction, so a transaction that includes multiple scan sessions can accumulate a large number of locks.

Lock promotion cannot occur if another task holds locks that conflict with the type of table lock 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 lock promotion is denied due to conflicting locks, a process can accumulate page or row locks in excess of the lock promotion threshold and may exhaust all available locks in Adaptive Server.

The lock promotion parameters are:

- For allpages-locked tables and datapages-locked tables, **page lock promotion HWM**, **page lock promotion LWM**, and **page lock promotion PCT.**
- For datarows-locked tables, **row lock promotion HWM**, **row lock promotion LWM**, and **row lock promotion PCT.**

The abbreviations in these parameters are:

- HWM, high water mark
- LWM, low water mark
- PCT, percent

The Lock Promotion High Water Mark

page lock promotion HWM and **row lock promotion HWM** set a maximum number of page or row locks allowed on a table before Adaptive Server attempts to escalate 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 the high water mark 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 the high water mark to less than 200 increases the chances of a particular task or worker process acquiring a table lock.

The Lock Promotion Low Water Mark

page lock promotion LWM and **row lock promotion LWM** set 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 the low water mark. The low water mark must be less than or equal to the corresponding high water mark.

Setting the low water mark to a very high value decreases the chance for a particular task or worker process to acquire a table lock, which uses more locks for the duration of the transaction, potentially exhausting all available locks in Adaptive Server. This possibility is especially high with queries that update a large number of rows in a datarows-locked table, or select large numbers of rows from datarows-locked tables at isolation levels 2 or 3. If conflicting locks prevent lock promotion, you may need to increase the value of the **number of locks** configuration parameter.

The Lock Promotion Percent

page lock promotion PCT and **row lock promotion PCT** set the percentage of locked pages or rows (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 or row locks to a table lock when the number of locks on the table exceeds:

```
(PCT * number of pages or rows 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 29-1 shows how Adaptive Server determines whether to promote page locks on a table to a table lock.

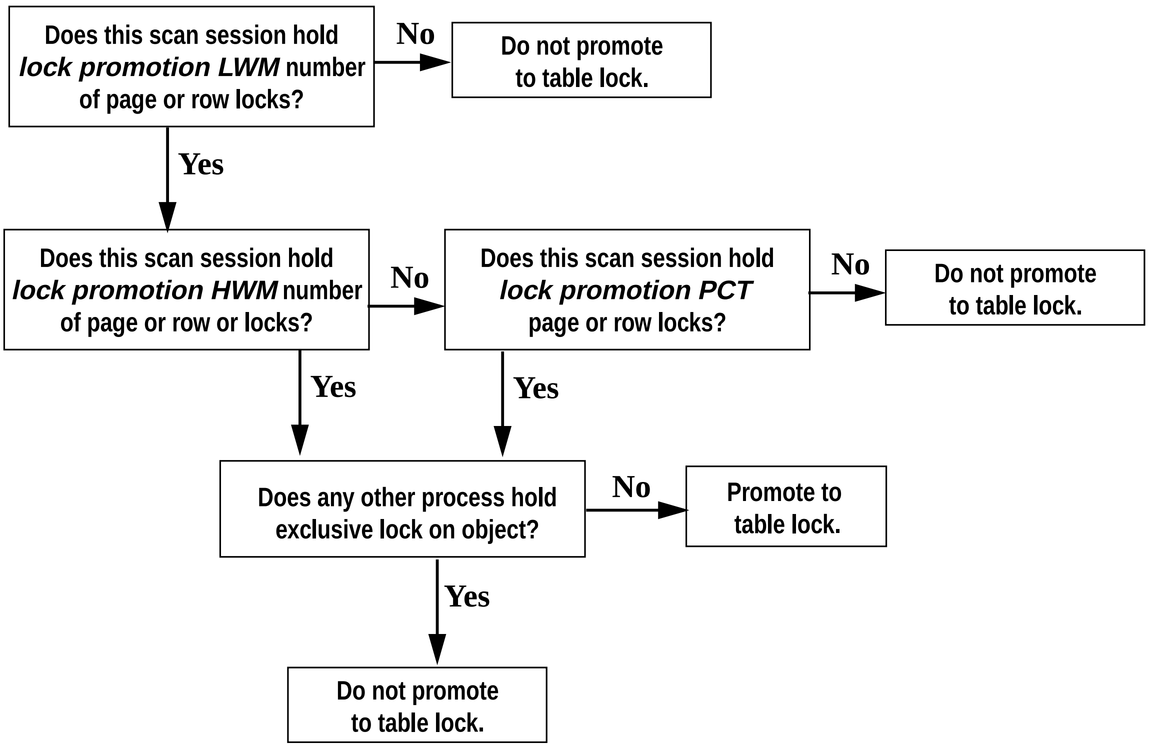


Figure 29-1: Lock promotion logic

Setting Server-Wide Lock Promotion Thresholds

The following command sets the server-wide **page lock promotion LWM** to 100, the **page lock promotion HWM** to 2000, and the **page lock promotion PCT** to 50 for all datapages-locked and allpages-locked tables:

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

In this example, the task does not attempt to promote to 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.

sp_setrowlockpromote sets the configuration parameters for all datarows-locked tables:

```
sp_setrowlockpromote "server", null, 300, 500, 50
```

The default values for lock promotion configuration parameters are likely to be appropriate for most applications.

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
```

```
sp_setrowlockpromote "table", authors, 300, 500, 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
```

```
sp_setrowlockpromote "table", authors, null, null,
50
```

To configure values for a database, use

```
sp_setpglockpromote "database", pubs3, 1000, 1100,
45
```

```
sp_setrowlockpromote "database", pubs3, 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 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 **sp_dropglockpromote** or **sp_droprowlockpromote**. When you drop a database's lock promotion thresholds, tables that do not have lock promotion thresholds configured 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 **sp_sysmon** to see how many times lock promotions take place and the types of promotions they are. See “Lock Promotions” on page 39-70 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 39-66 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.

Choosing the Locking Scheme for a Table

In general, choice of lock scheme for a new table should be determined by the likelihood that applications will experience lock contention on the table. The decision about whether to change the locking scheme for an existing table can be based on contention measurements on the table, but also needs to take application performance into account. Here are some typical situations and general guidelines for choosing the locking scheme:

- Applications require clustered access to the data rows due to range queries or **order by** clauses

  Allpages locking provides more efficient clustered access than data-only-locking.

- A large number of applications access about 10 to 20% of the data rows, with many updates and selects on the same data.

  Use datarows or datapages locking to reduce contention, especially on the tables with the highest contention.

- The table is a heap table that will have a high rate of inserts.

  Use datarows locking to avoid contention. If the number of rows inserted per batch is high, datapages locking is also acceptable. Allpages locking has more contention for the “last page” of heap tables.

- Applications need to maintain an extremely high transaction rate; contention is likely to be low.

  Use allpages locking; less locking and latching overhead yields improved performance.

Analyzing Existing Applications

If your existing applications experience blocking and deadlock problems, follow the steps below to analyze the problem:

1. Check for deadlocks and lock contention:
   - Use **sp_object_stats** to determine the tables where blocking is a problem.
   - Identify the table(s) involved in the deadlock, either using **sp_object_stats** or by enabling the configuration parameter **print deadlock information.**
2. If the table uses allpages locking and has a clustered index, ensure that performance of the modified clustered index

structure on data-only-locked tables will not hurt performance. See “Tables Where Clustered Index Performance Must Remain High” on page 29-17.

3. If the table uses allpages locking, convert the locking scheme to datapages locking to determine whether it solves the concurrency problem.
4. Rerun your concurrency tests. If concurrency is still an issue, change the locking scheme to datarows locking.

Choosing a Locking Scheme Based on Contention Statistics

If the locking scheme for the table is allpages, the lock statistics reported by **sp_object_stats** include both data page and index lock contention. If lock contention totals 15% or more for all shared, update, and exclusive locks, **sp_object_stats** recommends changing to datapages locking. You should make the recommended change, and run **sp_object_stats** again. If contention using datapages locking is more than 15%, **sp_object_stats** recommends changing to datarows locking. This two-phase approach is based on these characteristics:

- Changing from allpages locking to either data-only-locking scheme is time consuming and expensive, in terms of I/O cost, but changing between the two data-only-locking schemes is fast and does not require copying the table.
- Datarows locking requires more locks, and consumes more locking overhead. If your applications experience little contention after you convert high-contending tables to use datapages locking, you do not need to incur the locking overhead of datarows locking.

➤ ***Note***

The number of locks available to all processes on the server is limited by the configuration parameter **number of locks**. Changing to datapages locking reduces the number of locks required, since index pages are no longer locked. Changing to datarows locking can increase the number of locks required, since a lock is needed for each row. See “Estimating number of locks for Data-Only-Locked Tables” on page 29-6 for more information.

When examining **sp_object_stats** output, look at tables that are used together in transactions in your applications. Locking on tables that are used together in queries and transactions can affect the locking contention of the other tables. Reducing lock contention on one table could ease lock contention on other tables as well, or it could increase lock contention on another table that was masked by blocking on the first table in the application. For example:

- Lock contention is high for two tables that are updated in transactions involving several tables. Applications first lock *TableA*, then attempt to acquire locks on *TableB*, and block, holding locks on *TableA*. Additional tasks running the same application block while trying to acquire locks on *TableA*. Both tables show high contention and high wait times.

  Changing *TableB* to data-only locking may alleviate the contention on both tables.

- Contention for *TableT* is high, so its locking scheme is changed to a data-only locking scheme. Rerunning **sp_object_stats** now shows contention on *TableX*, which had shown very little lock contention. The contention on *TableX* was masked by the blocking problem on *TableT*.

If your application uses many tables, you may want to convert your set of tables to data-only locking gradually, by changing just those tables with the highest lock contention. Then test the results of these changes by rerunning **sp_object_stats**. You should run your usual performance monitoring tests both before and after the changes are made.

Monitoring and Managing Tables After Conversion

After you have converted one or more tables in an application to a data-only-locking scheme:

- Check query plans and I/O statistics, especially for those queries that use clustered indexes.
- Monitor the tables to learn how changing the locking scheme affects:
  - The cluster ratios, especially for tables with clustered indexes
  - The number of forwarded rows in the table

Applications Not Likely to Benefit from Data-Only Locking

This section describes tables and application types that may get little benefit from converting to data-only locking or may require additional management after the conversion.

Tables Where Clustered Index Performance Must Remain High

If queries with high performance requirements use clustered indexes to return large numbers of rows in index order, you may see performance degradation if you change these tables to use data-only locking. Clustered indexes on data-only-locked tables are structurally the same as nonclustered indexes. Placement algorithms keep newly inserted rows close to existing rows with adjacent values, as long as space is available on nearby pages.

Performance for a data-only-locked table with a clustered index should be close to the performance of the same table with allpages locking immediately after a **create clustered index** command or a **reorg rebuild** command, but performance, especially with large I/O, declines if cluster ratios decline because of inserts and forwarded rows.

Performance remains high for tables that do not experience a lot of inserts. On tables that get a lot of inserts, a System Administrator may need to drop and re-create the clustered index or run **reorg rebuild** more frequently. Using space management properties such as **fillfactor**, **exp_row_size**, and **reservepagegap** can help reduce the frequency of maintenance operations. In some cases, using the allpages locking scheme for the table, even if there is some contention, may provide better performance for queries performing clustered index scans than using data-only locking for the tables.

Tables with Maximum-Length Rows

Data-only-locked tables require more overhead per page and per row than allpages-locked tables, so the maximum row size for a data-only-locked table is slightly shorter than the maximum row size for an allpages-locked table. For tables with fixed-length columns only, the maximum row size is 1958 bytes of user data for data-only-locked tables. Allpages-locked tables allow a maximum of 1960 bytes.

For tables with variable-length column, subtract 2 bytes for each variable-length columns (this includes all columns that allow null values). For example, the maximum user row size for a data-only-locked table with 4 variable-length columns is 1950 bytes.

If you try to convert an allpages-locked table that has more than 1958 bytes in fixed-length columns, the command fails as soon as it reads the table schema. When you try to convert an allpages-locked table with variable-length columns, and some rows exceed the maximum size for the data-only-locked table, the **alter table** command fails at the first row that is too long to convert.

Application Maintenance

30 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:

The common-sense approach is to perform maintenance tasks, whenever possible, when your Adaptive Server usage is low. This chapter can help you determine what kind of performance impacts these maintenance activities have on applications and on overall Adaptive Server performance.

Running *reorg* on Tables and Indexes

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

- **reclaim_space** – Clears committed deletes and space left when updates shorten the length of data rows.
- **forwarded_rows** – Returns forwarded rows to home pages.
- **compact** – Performs both of the operations above.
- **rebuild** – Rebuilds an entire table or index.

When you run **reorg rebuild** on a table, it locks the table for the entire time it takes to rebuild the table and its indexes. All of the other **reorg** commands, including **reorg rebuild** on an index, lock a small number of pages at a time, and use short, independent transactions to perform their work. This means that you should schedule the **reorg rebuild** command on a table when users do not need access to the table. The other **reorg** commands can be run at any time. The only negative effects might be on systems that are very I/O bound.

For more information on running reorg commands, see Chapter 24, “Using the reorg Command,” in the *System Administration Guide*.

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 parameter **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. See “Configuring Resources for Parallel Sorting” on page 13-6 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 that you have not dumped since you created the index, the entire **create index** process is executed again while loading transaction log dumps.

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

Creating an Index on Sorted Data

If you need to re-create a clustered index or create one on data that was bulk copied into the server in index key order, use the **sorted_data** option to **create index** to shorten index creation time.

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

When creating an index on a nonpartitioned table, **sorted_data** and the use of any of the following clauses requires 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 30-1: Using options for creating a clustered index

Options	Partitioned Table	Unpartitioned Table
No options specified	Parallel sort; copies data, distributing evenly on partitions; creates index tree.	Either parallel or nonparallel sort; copies data, creates index tree.
with sorted_data only or **with sorted_data on** *same_segment*	Creates index tree only. Does not perform the sort or copy data. Does not run in parallel.	Creates index tree only. Does not perform the sort or copy data. Does not run in parallel.
with sorted_data and **ignore_dup_row** or **fillfactor** or **on** *other_segment* or **max_rows_per_page**	Parallel sort; copies data, distributing evenly on partitions; creates index tree.	Copies data and creates the index tree. Does not perform the sort. Does not run in parallel.

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

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

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

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

Creating or Altering a Database

Creating or altering a database is I/O intensive; consequently, other I/O-intensive operations may suffer. When you create a database, Adaptive Server copies the *model* database to the new database and then initializes all the allocation pages and clears database pages.

The following procedures can speed database creation or minimize its impact on other processes:

- Use the **for load** option to **create database** if you are restoring a database, that is, if you are getting ready to issue a **load database** command.

  When you create a database without **for load**, it copies *model* and then initializes all of the allocation units. When you use **for load**, it postpones zeroing the allocation units until the load is complete. Then it initializes only the untouched allocation units. If you are loading a very large database dump, this can save a lot of time.

- Create databases during off-hours if possible.

create database and **alter database** perform concurrent parallel I/O when clearing database pages. The number of devices is limited by the configuration parameter **number of large i/o buffers**. The default value for this parameter is 6, allowing parallel I/O on six devices at once. A single **create database** and **alter database** can use up to eight 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. As the I/O to each device completes, the 16K buffers are used for remaining devices listed in the command. The following example names 10 separate devices:

```
create database hugedb
      on dev1 = 100,
      dev2 = 100,
      dev3 = 100,
      dev4 = 100,
      dev5 = 100,
      dev6 = 100,
      dev7 = 100,
      dev8 = 100
log on logdev1 = 100,
      logdev2 = 100
```

During operations that use these buffers, a message is sent to the log when the number of buffers is exceeded. This information for the **create database** command above shows that **create database** started clearing devices on the first 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'
01:00000:00013:1999/07/26 15:36:17.54 server  No disk i/o buffers are
available for this operation. The total number of buffers is
controlled by the configuration parameter 'number of large i/o buffers'.
CREATE DATABASE: allocating 51200 pages on disk 'dev7'
CREATE DATABASE: allocating 51200 pages on disk 'dev8'
CREATE DATABASE: allocating 51200  pages on disk 'logdev1'
CREATE DATABASE: allocating 51200  pages on disk 'logdev2'
```

When **create database** copies *model*, it uses 2K I/O.

See “number of large i/o buffers” on page 17-23 of the *System Administration Guide*.

Backup and Recovery

All Adaptive Server backups are performed by a Backup Server. The backup architecture uses a client/server paradigm, with Adaptive Servers as clients to a Backup Server.

Local Backups

Adaptive Server sends the local Backup Server instructions, via remote procedure calls, telling the Backup Server which pages to dump or load, which backup devices to use, and other options. Backup Server performs all the disk I/O. Adaptive Server does not read or send dump and load data, 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 26, “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 32-32.

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 copying into a table on Adaptive Server runs fastest when there are no indexes or active triggers on the table. When you are running fast bulk copy, Adaptive Server performs reduced logging. It does not log the actual changes to the database, only the allocation of pages. And, since there are no indexes to update, it saves all the time it would otherwise take to update indexes for each data insert and to log the changes to the index pages.

To use fast bulk copy:

- Drop any indexes; re-create them when the bulk copy completes.
- Use **alter table...disable trigger** to deactivate triggers during the copy; use **alter table...enable trigger** after the copy completes.
- Set the **select into/bulkcopy/pllsort** option with **sp_dboption.** Remember to turn the option off after the bulk copy operation completes.

During fast bulk copy, rules are not enforced, but defaults **are** enforced.

Since changes to the data are not logged, you should perform a **dump database** soon after a fast bulk copy operation. Performing a fast bulk copy in a database blocks the use of **dump transaction**, since the unlogged data changes cannot be recovered from the transaction log dump.

Parallel Bulk Copy

For fastest performance, you can use fast bulk copy to copy data into partitioned tables. For each bulk copy session, you specify the partition on which the data should reside. If your input file is already in sorted order, you can bulk copy data into partitions in order, and avoid the sorting step while creating clustered indexes. See "Steps for Partitioning Tables" on page 33-32 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.
- Set the **number of preallocated extents** configuration parameter high if you perform many large bulk copies. See “number of pre-allocated extents” on page 17-141 of the *System Administration Guide*.
- Find the optimal network packet size. See “Client Commands for Larger Packet Sizes” on page 36-5.

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 33-32 for more information on using bulk copy with partitioned tables.

Adding Large Amounts of Data to a Table

When you are adding 10 to 20 percent or more to a large table, drop the nonclustered indexes, load the data, and then re-create nonclustered indexes.

For very large tables, you may need to leave the clustered index in place due to space constraints. Adaptive Server must make a copy of the table when it creates a clustered index. In many cases, once tables become very large, the time required to perform a slow bulk copy with the index in place is less than the time to perform a fast bulk copy and re-create the clustered index.

Using Partitions and Multiple Bulk Copy Processes

If you are loading data into a table without indexes, you can create partitions on the table and use one **bcp** session for each partition. See "Using Parallel bcp to Copy Data into Partitions" on page 33-25.

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 25-19 of the *System Administration Guide* for information about **dbcc** and locking. Also see “Scheduling Database Maintenance at Your Site” on page 25-21 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)
```

The final clean up 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)
```

Determining the Space Available for Maintenance Activities

Several maintenance operations require room to make a copy of the data pages of a table:

- **create clustered index**
- **alter table...lock**
- Some **alter table** commands that add or modify columns
- **reorg rebuild** on a table

In most cases, these commands also require space to re-create any indexes, so you need to determine:

- The size of the table and its indexes
- The amount of space available on the segment where the table is stored

- The space management properties set for the table and its indexes

The following sections describe tools that provide information on space usage and space availability.

Overview of Space Requirements

Any command that copies a table's rows also re-creates all of the indexes on the table. You need space for a complete copy of the table and copies of all indexes. These commands do not estimate how much space is needed. They stop with an error message if they run out of space on any segment used by the table or its indexes. For large tables, this could occur minutes or even hours after the command starts.

You need free space on the segments used by the table and its indexes, as follows:

- Free space on the table's segment must be at least equal to:
  - The size of the table, plus
  - Approximately 20 percent of the table size, if the table has a clustered index and you are changing from allpages locking to data-only locking
- Free space on the segments used by nonclustered indexes must be at least equal to the size of the indexes.

Clustered indexes for data-only-locked tables have a leaf level above the data pages. If you are altering a table with a clustered index from allpages locking to data-only locking, the resulting clustered index requires more space. The additional space required depends on the size of the index keys.

Tools for Checking Space Usage and Space Available

As a simple guideline, copying a table and its indexes requires space equal to the current space used by the table and its indexes, plus about 20% additional room. However:

- If data modifications have created many partially-full pages, space required for the copy of the table can be smaller than the current size.
- If space-management properties for the table have changed, or if space required by **fillfactor** or **reservepagegap** has been filled by data

modifications, the size required for the copy of the table can be larger.

- Adding columns or modifying columns to larger datatypes requires more space for the copy.

Note that log space is also required.

Checking Space Used for Tables and Indexes

To see the size of a table and its indexes, use **sp_spaceused**, as follows:

```
sp_spaceused titles, 1
```

If the table has a clustered index, and you are converting from allpages locking to data-only locking, see "Calculating the Sizes of Data-Only-Locked Tables" on page 15-18 for information on estimating the size of the clustered index.

Checking Space on Segments

Tables are always copied to free space on the segment where they are currently stored, and indexes are re-created on the segment where they are currently stored. Commands that create clustered indexes can specify a segment. The copy of the table and the clustered index are created on the target segment.

To determine the number of pages available on a segment, use **sp_helpsegment**. The last line of **sp_helpsegment** shows the total number of free pages available on a segment.

The following command prints segment information for the *default* segment, where objects are stored when no segment was explicitly specified:

```
sp_helpsegment "default"
```

sp_helpsegment reports the names of indexes on the segment. If you do not know the segment name for a table, use **sp_help** and the table name. The segment names for indexes are also reported by **sp_help**.

Checking Space Requirements for Space Management Properties

If you make significant changes to space management property values, the table copy can be considerably larger or smaller than the original table. Settings for space management properties are stored in the *sysindexes* tables, and are displayed by **sp_help** and **sp_helpindex**.

This output shows the space management properties for the *titles* table:

```
exp_row_size reservepagegap fillfactor max_rows_per_page
------------ -------------- ---------- -----------------
         190             16         90                 0
```

sp_helpindex produces this report:

```
index_name          index_description
    index_keys
    index_max_rows_per_page index_fillfactor index_reservepagegap
    ----------------------- ---------------- --------------------
title_id_ix         nonclustered located on default
    title_id
                         0               75                    0
title_ix            nonclustered located on default
    title
                         0               80                   16
type_price          nonclustered located on default
    type, price
                         0               90                    0
```

Space Management Properties Applied to the Table

During the copy step, the space management properties for the table are used, as follows:

- If an expected row size value is specified for the table, and the locking scheme is being changed from allpages locking to data-only-locking, the expected row size is applied to the data rows as they are copied. If no expected row size is set, but there is a **max_rows_per_page** value for the table, an expected row size is computed, and that value is used. Otherwise, the default value specified with the configuration parameter **default exp_row_size percent** is used for each page allocated for the table.
- The **reservepagegap** is applied as extents are allocated to the table.
- If **sp_chgattribute** has been used to save a **fillfactor** value for the table, it is applied to the new data pages as the rows are copied.

Space Management Properties Applied to the Index

When the indexes are rebuilt, space management properties for the indexes are applied, as follows:

- If **sp_chgattribute** has been used to save **fillfactor** values for indexes, these values are applied when the indexes are re-created.

- If **reservepagegap** values are set for indexes, these value are applied when the indexes are re-created.

Estimating the Effects of Space Management Properties

Table 30-2 shows how to estimate the effects of setting space management properties.

Table 30-2: Effects of space management properties on space use

Property	Formula	Example
fillfactor	Requires (100 / **fillfactor**) * *num_pages* if pages are currently fully packed	**fillfactor** of 75 requires 1.33 times current number of pages; a table of 1,000 pages grows to 1,333 pages.
reservepagegap	Increases space by 1 / **reservepagegap** if extents are currently filled	**reservepagegap** of 10 increase space used by 10%; a table of 1,000 pages grows to 1,100 pages.
max_rows_per_page	Converted to **exp_row_size** when converting to data-only-locking	See Table 30-3 on page 30-15.
exp_row_size	Increase depends on number of rows smaller than *exp_rowsize*, and the average length of those rows	If **exp_row_size** is 100, and 1,000 rows have a length of 60, the increase in space is: (100 - 60) * 1000 or 40,000 bytes; approximately 20 additional pages.

For more information, see “Setting Space Management Properties” on page 31-1.

If a table has **max_rows_per_page** set, and the table is converted from allpages locking to data-only locking, the value is converted to an **exp_row_size** value before the **alter table...lock** command copies the table to its new location. The **exp_row_size** is enforced during the copy. Table 30-3 shows how the values are converted.

Table 30-3: Converting max_rows_per_page to exp_row_size

If *max_rows_per_page* Is Set To	Set *exp_row_size* To
0	Percentage value set by **default exp_row_size percent**
255	1, that is, fully packed pages
1–254	The smaller of: • maximum row size • 2002 / **max_rows_per_page** value

If There Is Not Enough Space

If there is not enough space to copy the table and re-create all the indexes, determine whether dropping the nonclustered indexes on the table leaves enough room to create a copy of the table. Without any nonclustered indexes, the copy operation requires space just for the table and the clustered index. Do not drop the clustered index, since it is used to order the copied rows, and attempting to re-create it later may require space to make a copy of the table. Re-create the nonclustered indexes after the command completes.

31 Setting Space Management Properties

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

This chapter contains the following sections:

Reducing Index Maintenance with *fillfactor*

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. When you use **fillfactor**, except for a fillfactor value of 100 percent, data and index rows use more disk space than the default setting requires.

If you are creating indexes for tables that will grow in size, you can reduce the impact of page splitting on your tables and indexes by using the **fillfactor** option for **create index**. 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 experience growth; maintaining that fillfactor by continuing to split partially full pages defeats the purpose.

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

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

fillfactor values specified with **create index** are applied at the time the index is created. They are not saved in *sysindexes*, and the fullness of the data or index pages is not maintained over time.

You can also use **sp_chgattribute** to store values for **fillfactor** that are used when **reorg rebuild** is run on a table. See "Setting fillfactor Values with sp_chgattribute" on page 31-3 for more information.

Advantages of Using *fillfactor*

Setting **fillfactor** to a low value provides a temporary performance enhancement. Its benefits fade as inserts to the database increase the amount of space used on data or index pages. A lower fillfactor provides these benefits:

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

Disadvantages of Using *fillfactor*

If you use **fillfactor**, especially a very low **fillfactor**, you may notice these effects on queries and maintenance activities:

- More pages must be read for each query that does a table scan or leaf-level scan on a nonclustered index. In some cases, it may also add a level to an index's B-tree structure, since there will be more pages at the data level and possibly more pages at each index level.
- **dbcc** commands need to check more pages, so **dbcc** commands take more time.
- **dump database** time increases, because more pages need to be dumped. **dump database** copies all pages that store data, but does not dump pages that are not yet in use. Your dumps and loads will take longer to complete and may use more tapes.

- Fillfactors fade away over time. If you use **fillfactor** to reduce the performance impact of page splits, you need to monitor your system and re-create indexes when page splitting begins to hurt performance.

Setting *fillfactor* Values with *sp_chgattribute*

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

- When you run **reorg rebuild** to restore the cluster ratios of data-only-locked tables and indexes
- When you use **alter table...lock** to change the locking scheme for a table or you use an **alter table...add/modify** command that requires copying the table
- When you run **create clustered index** and there is a value stored for the table

The stored **fillfactor** is not applied when nonclustered indexes are rebuilt as a result of a **create clustered index** command. Creating a clustered index follows the behavior of pre-11.9.2 releases:

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

fillfactor Examples

The following examples show the application of **fillfactor** values.

No Stored *fillfactor* Values

With no **fillfactor** values stored in *sysindexes*, the **fillfactor** specified in commands that create indexes are applied as shown in Table 31-1.

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

Table 31-1: fillfactor values applied with no table-level saved value

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

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

Values Used for alter table...lock and reorg rebuild

When no **fillfactor** values are stored, both **alter table...lock** and **reorg rebuild** apply the server-wide default value, set by the configuration parameter **default fill factor percentage**. The default **fillfactor** is applied as shown in Table 31-2.

Table 31-2: fillfactor values applied with during rebuilds

Command	Allpages-Locked Table	Data-Only-Locked Table
Clustered index rebuild	Data pages: default value	Data pages: fully packed Leaf pages: default value
Nonclustered index rebuilds	Leaf pages: default	Leaf pages: default

Table-Level or Clustered Index *fillfactor* Value Stored

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

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

With 50 as the stored table-level value for **fillfactor**, the following create **clustered index command** applies the **fillfactor** values shown in Table 31-3.

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

Table 31-3: Using stored fillfactor values for clustered indexes

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

➤ ***Note***

When a **create clustered index** command is run, any table-level **fillfactor** value stored in *sysindexes* is reset to 0. To affect the filling of data-only-locked data pages during a **create clustered index** or **reorg** command, you must issue **sp_chgattribute** before running the command.

Effects of alter table...lock When Values Are Stored

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

Tables with Clustered Indexes

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

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

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

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

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

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

Table 31-4 shows the fillfactors used on data and index pages by an **alter table...lock** command, executed after the **sp_chgattribute** commands above have been run.

Table 31-4: Effects of stored fillfactor values during alter table

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

➤ ***Note***

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

fillfactor Values Stored for Nonclustered Indexes

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

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

Table 31-5 shows the effects of a **reorg rebuild** command on a data-only-locked table when the **sp_chgattribute** commands above are used to store **fillfactor** values.

Table 31-5: Effect of stored fillfactor values during reorg rebuild

reorg rebuild	No Clustered Index	Clustered Index	Nonclustered Indexes
Data-only-locked table	Data pages: 80	Data pages: 50 Leaf pages: 80	*ncl_ix* leaf pages: 90 *pubid_ix* leaf pages: 75

Use of the *sorted_data* and *fillfactor* Options

The **sorted_data** option for **create index** is used when the data to be sorted is already in order by the index key. This allows **create clustered index** to skip the copy step while creating a clustered index. For example, if data that is bulk copied into a table is already in order by the clustered index key, creating an index with the **sorted_data** option creates the index without performing a sort. If the data does not need to be copied to new pages, the **fillfactor** is not applied.

The use of other **create index** options might still require copying, however. For more information, see“Creating an Index on Sorted Data” on page 30-3.

Reducing Row Forwarding with Expected Row Size

Specifying an expected row size for a data-only-locked table is useful when an application allows rows that contain null values or short variable-length character fields to be inserted, and these rows grow in length with subsequent updates. The major purpose of setting an expected row size is to reduce row forwarding.

For example, the *titles* table in the *pubs2* database has many *varchar* columns and columns that allow null values. The maximum row size for this table is 331 bytes, and the average row size (as reported by **optdiag**) is 184 bytes, but it is possible to insert a row with less than 40 bytes, since many columns allow null values. In a data-only-locked table, inserting short rows and then updating them can result in row forwarding. See “Updates on Data-Only-Locked Heap Tables” on page 3-18 for more information.

You can set the expected row size for tables with variable-length columns, as follows:

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

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

Default, Minimum, and Maximum Values for *exp_row_size*

Table 31-6 shows the minimum and maximum values for expected row size and the meaning of the special values 0 and 1.

Table 31-6: Valid values for expected row size

exp_row_size Values	Minimum, Maximum, and Special Values
Minimum	The greater of: • 2 bytes • The sum of all fixed-length columns
Maximum	Maximum data row length
0	Use server-wide default value
1	Fully pack all pages; do not reserve room for expanding rows

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

Default Value

If you do not specify an expected row size or a value of 0 when you create a data-only-locked table with variable-length columns, Adaptive Server uses the amount of space specified by the configuration parameter **default exp_row_size percent** for any table that has variable-length columns. See "Setting a Default Expected Row Size Server-Wide" on page 31-10 for information on how this parameter affects space on data pages.

Minimum Value and Maximum Value

The minimum value for expected row size is the greater of:

- The minimum row size, 2 bytes
- The size of all the fixed-length columns for the table

The maximum value is the maximum data row length for the table.

Use **sp_help** to see the defined length of the columns in the table.

Specifying Fully Packed Pages

If you do not want space on a page to be reserved for expanding updates, you can specify a value of 1 for the expected row size, which creates fully packed pages.

Specifying an Expected Row Size with *create table*

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

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

Adding or Changing an Expected Row Size with *sp_chgattribute*

To add or change the expected row size for a table, us **sp_chgattribute**. This sets the expected row size to 190 for the *new_titles* table:

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

If you want a table to switch to the **default exp_row_size percent** instead of a current, explicit value, enter:

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

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

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

- When a clustered index on the table is created or **reorg rebuild** is run on the table. The expected row size is applied as rows are copied to new data pages. If you increase **exp_row_size**, and re-create the clustered index or run **reorg rebuild**, the new copy of the table may require more storage space.
- The next time a page is affected by data modifications.

Setting a Default Expected Row Size Server-Wide

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

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

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

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

Displaying the Expected Row Size for a Table

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

```
sp_help titles
```

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

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

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

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

Choosing an Expected Row Size for a Table

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

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

Using *optdiag* to Check for Forwarded Rows

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

```
Statistics for table:                "titles"

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

You can also use **optdiag** to check the number of forwarded rows for a table to determine whether your setting for **exp_row_size** is reducing the number of forwarded rows generated by your applications. For more information on **optdiag**, see Chapter 19, "Statistics Tables and Displaying Statistics with optdiag."

Querying *systabstats* to Check for Forwarded Rows

You can check the *forwrowcnt* column in the *systabstats* table to see the number of forwarded rows for a table. This query checks the number of forwarded rows for all user tables (those with object IDs greater than 100):

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

➤ ***Note***

Forwarded row counts are updated in memory, and the housekeeper periodically flushes them to disk. If you need to query the *systabstats* table using SQL, use **sp_flushstats** first to ensure that the most recent statistics are available. **optdiag** flushes statistics to disk before displaying values.

Conversion of *max_rows_per_page* to *exp_row_size*

If a **max_rows_per_page** value is set for an allpages-locked table, the value is used to compute an expected row size during the **alter table...lock** command. The formula is shown in Table 31-7.

Table 31-7: Conversion of max_rows_per_page to exp_row_size

Value of *max_rows_per_page*	Value of *exp_row_size*
0	Percentage value set by **default exp_row_size percent**
255	1 (fully packed pages)
1-254	The smaller of: • Maximum row size • 2002 / **max_rows_per_page** value

For example, if **max_rows_per_page** is set to 10 for an allpages-locked table with a maximum defined row size of 300 bytes, the **exp_row_size** value will be 200 (2002 / 10) after the table is altered to use data-only locking. If **max_rows_per_page** is set to 10, but the maximum defined row size is only 150, the expected row size value will be set to 150.

Monitoring and Managing Tables That Use Expected Row Size

After setting an expected row size for a table, use **optdiag** or queries on *systabstats* to check the number of forwarded rows still being generated by your applications. Run **reorg forwarded_rows** if you feel that the number of forwarded rows is high enough to affect application performance. The **reorg forwarded_rows** command uses short transactions and is very nonintrusive, so it does not cause performance problems if it is run while applications are active. See "Moving Forwarded Rows to Home Pages" on page 24-3 of the *System Administration Guide* for more information.

If the application still results in a large number of forwarded rows, you may want to increase the expected row size for the table, using **sp_chgattribute**.

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

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

Leaving Space for Forwarded Rows and Inserts

Setting a **reservepagegap** value can reduce the frequency of maintenance activities such as running **reorg rebuild** and re-creating indexes for some tables to maintain high performance. Good performance on data-only-locked tables requires good data clustering on the pages, extents, and allocation units used by the table.

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

Row and page cluster ratios are usually 1.0, or very close to 1.0, immediately after a clustered index is created on a table or immediately after **reorg rebuild** is run. However, future data modifications can cause row forwarding and can require allocation of additional data and index pages to store inserted rows. Setting a reserve page gap can reduce storage fragmentation and reduce the frequency with which you need to re-create indexes or run **reorg rebuild** on the table.

Extent Allocation Operations and *reservepagegap*

Commands that allocate many data pages perform **extent allocation** to allocate eight pages at a time, rather than allocating just one page at a time. Extent allocation reduces logging, since it writes one log record instead of eight.

Commands that perform extent allocation are: **select into**, **create index**, **reorg rebuild**, **bcp**, **alter table...lock**, and the **alter table...unique** and **primary key** constraint options, since these constraints create indexes. **alter table** commands that add, drop, or modify columns sometimes require a table-copy operation also. All of these commands allocate an extent, and, unless a reserve page gap value is in effect, fill all eight pages.

You specify the **reservepagegap** in pages, indicating a ratio of empty pages to filled pages. For example, if you specify a **reservepagegap** of 8, an operation doing extent allocation fills 7 pages and leaves the eighth page empty.

These empty pages can be used to store forwarded rows and for maintaining the clustering of data rows in index key order, for data-only-locked tables with clustered indexes.

Since extent allocation operations must allocate entire extents, they do not use the first page on each allocation unit, because it stores the allocation page. For example, if you create a clustered index on a large table and do not specify a reserve page gap, each allocation unit has 7 empty, unallocated pages, 248 used pages, and the allocation page. These 7 pages can be used for row forwarding and inserts to the table, which helps keep forwarded rows and inserts with clustered indexes on the same allocation unit. Using **reservepagegap** leaves additional empty pages on each allocation unit.

Figure 31-1 shows how an allocation unit might look after a clustered index is created with a **reservepagegap** value of 16 on the table. The pages that share the first extent with the allocation unit are not used and are not allocated to the table. Pages 279, 295, and 311 are the unused pages on extents that are allocated to the table.

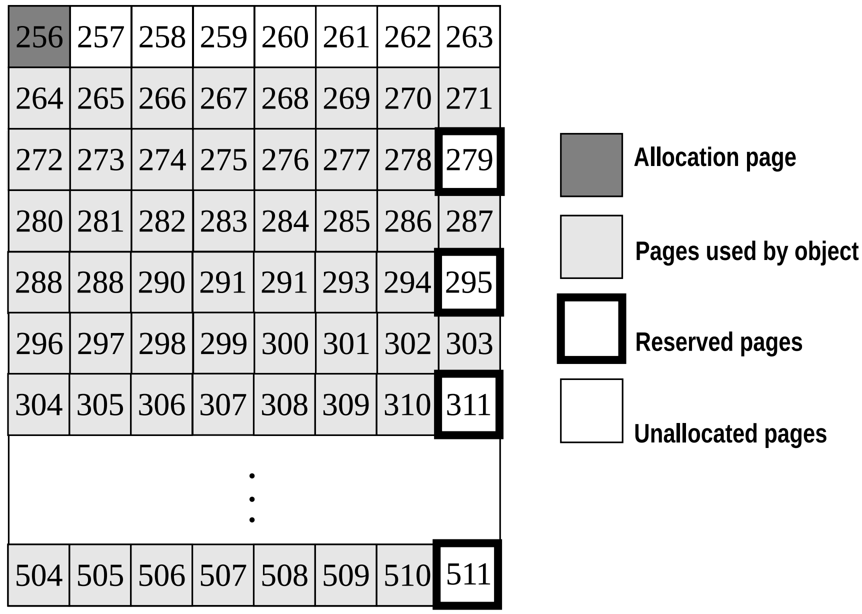


Figure 31-1: Reserved pages after creating a clustered index

Specifying a Reserve Page Gap with *create table*

This **create table** command specifies a **reservepagegap** value of 16:

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

Any operation that performs extent allocation on the *more_titles* table leaves 1 empty page for each 15 filled pages.

The default value for **reservepagegap** is 0, meaning that no space is reserved. The maximum value is 255, meaning that 1 page is left unused on each allocation unit.

Specifying a Reserve Page Gap with *create index*

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

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

You can also specify a **reservepagegap** value with the **alter table...constraint** options, **primary key** and **unique**, that create indexes. This example creates a unique constraint:

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

Changing *reservepagegap* with *sp_chgattribute*

The following command uses **sp_chgattribute** to change the reserve page gap for the *titles* table to 20:

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

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

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

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

- When data is bulk-copied into the table, the reserve page gap is applied to all newly allocated space, but the storage of existing pages is not affected.
- When the **reorg rebuild** command is run on the table, the reserve page gap is applied as the table is copied to new data pages.
- When a clustered index is created, the reserve page gap value stored for the table is applied to the data pages.

The reserve page gap is applied to index pages:

- During **alter table...lock**, while rebuilding indexes for the table

- During **reorg rebuild** commands that affect indexes
- During **create clustered index** and **alter table** commands that create a clustered index, as nonclustered indexes are rebuilt

reservepagegap Examples

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

reservepagegap Specified Only for the Table

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

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

Table 31-8 shows the values applied when running **reorg rebuild** or dropping and creating a clustered index.

Table 31-8: reservepagegap values applied with table-level saved value

Command	Allpages-Locked Table	Data-Only-Locked Table
create clustered index or clustered index rebuild due to **reorg rebuild**	Data and index pages: 16	Data pages: 16 Index pages: 0 (filled extents)
Nonclustered index rebuild	Index pages: 0 (filled extents)	Index pages: 0 (filled extents)

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

reservepagegap Specified for a Clustered Index

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

```
sp_chgattribute titles, "reservepagegap", 16
```

```
create clustered index title_ix on titles(title)
    with reservepagegap = 20

create index type_price on titles(type, price)
    with reservepagegap = 24
```

Table 31-9 shows the effects of this sequence of commands.

Table 31-9: reservepagegap values applied with for index pages

Command	Allpages-Locked Table	Data-Only-Locked Table
create clustered index or clustered index rebuild due to **reorg rebuild**	Data and index pages: 20	Data pages: 16 Index pages: 20
Nonclustered index rebuilds	Index pages: 24	Index pages: 24

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

Choosing a Value for *reservepagegap*

Choosing a value for **reservepagegap** depends on:

- Whether the table has a clustered index,
- The rate of inserts to the table,
- The number of forwarded rows that occur in the table, and
- How often you re-create the clustered index or run the **reorg rebuild** command.

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

Monitoring *reservepagegap* Settings

You can use **optdiag** to check the cluster ratio and the number of forwarded rows in tables. Declines in cluster ratios can also indicate that running **reorg** commands could improve performance:

- If the data page cluster ratio for a clustered index is low, run **reorg rebuild** or drop and re-create the clustered index.
- If the index page cluster ratio is low, drop and re-create the non-clustered index.

If you want to reduce the frequency with which you run **reorg** commands to maintain cluster ratios, increase the **reservepagegap** slightly before running **reorg rebuild**. See Chapter 19, "Statistics Tables and Displaying Statistics with optdiag," for more information on **optdiag**.

reservepagegap and *sorted_data* Options to *create index*

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

If you specify both options with **create clustered index**:

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

Background on the *sorted_data* Option

Besides **reservepagegap**, other options to **create clustered index** may require a sort, which causes the **sorted_data** option to be ignored. For more information, see "Creating an Index on Sorted Data" on page 30-3. In particular, the following comments relate to the use of **reservepagegap**:

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

Table 31-10 shows how these rules apply.

Table 31-10:reservepagegap and sorted_data options

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

Matching Options and Goals

If you want to redistribute the data pages of a table, leaving room for later expansion:

- For allpages-locked tables, drop and re-create the clustered index without using the **sorted_data** option. Specify the desired **reservepagegap** value in the **create clustered index** command, if the value stored in *sysindexes* is not the value you want.

- For data-only-locked tables, use **sp_chgattribute** to set the **reservepagegap** for the table to the desired value and then drop and re-create the clustered index, without using the **sorted_data** option. The **reservepagegap** stored for the table applies to the data pages. If **reservepagegap** is specified in the **create clustered index** command, it applies only to the index pages.

To create a clustered index without copying data pages:

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

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

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

Using *max_rows_per_page* on Allpages-Locked Tables

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

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

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

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

Reducing Lock Contention 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.

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

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

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

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

Indexes and *max_rows_per_page*

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

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

For nonclustered indexes, the maximum value for **max_rows_per_page** is the number of index rows that fit on the leaf page, 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

sp_chgattribute configures the **max_rows_per_page** of a table or an index. **sp_chgattribute** affects all future operations; it does not change existing pages. For example, to change the **max_rows_per_page** value of the *authors* table to 1, enter:

```
sp_chgattribute authors, "max_rows_per_page", 1
```

There are two ways to apply a **max_rows_per_page** value to existing data:

- If the table has a clustered index, drop and re-create the index with a **max_rows_per_page** value.
- Use the **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.

System-Level Tuning

32 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:

Chapter 14, "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 memor**y 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:

- 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 14, “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 32-1 shows how memory is divided.

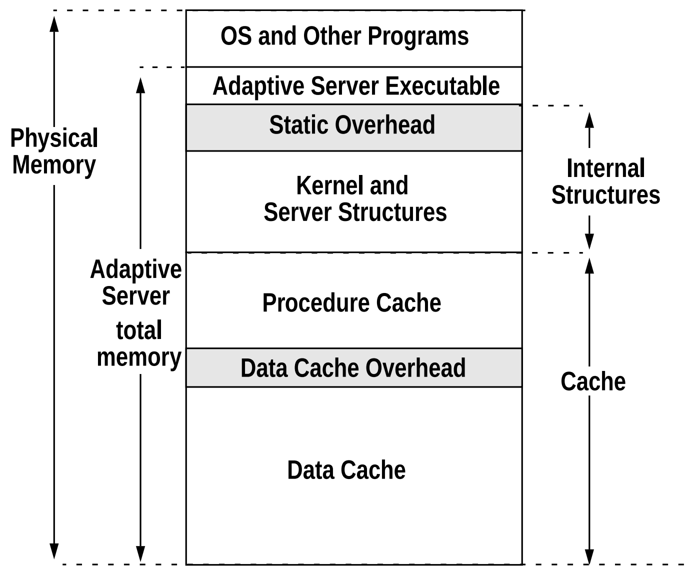


Figure 32-1: How Adaptive Server uses memory

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 stored procedures and triggers and for short-term memory needs such as statistics and query plans for parallel queries.
- The **data cache** – Used for data, index, and log pages. The data cache can be divided into separate, named caches, with specific databases or database objects bound to specific caches.

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 the **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.

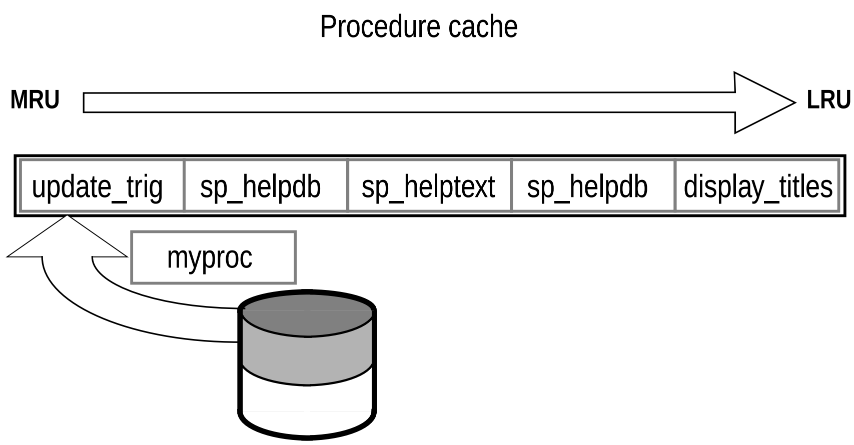


Figure 32-2: The procedure cache

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 32-3.

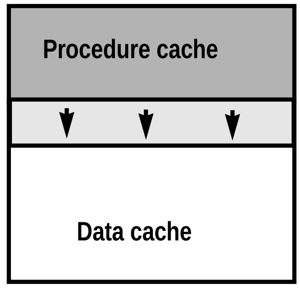


Figure 32-3: Effect of increasing procedure cache size on the data cache

When you first install Adaptive Server, the default procedure cache size is configured as 20% 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. The configuration parameter to set the size, **procedure cache percent**, is documented in Chapter 17, "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 32-4.

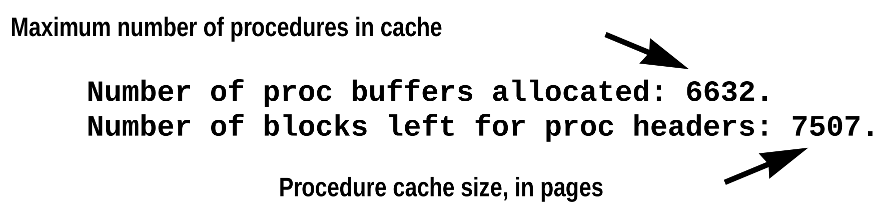


Figure 32-4: Procedure cache size messages in the error log

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 32-4, no more than 6632 compiled objects can reside in the procedure cache simultaneously.

proc headers

"proc headers" representshe number of 2K pages dedicated to the procedure cache. In Figure 32-4, 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 32-5 make a good starting point.

Procedure cache size = (Max # of concurrent users) * (Size of largest plan) * 1.25

Minimum procedure cache size needed = (# of main procedures) * (Average plan size)

Figure 32-5: 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 32-5.

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 39-87.

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
- Recently accessed data pages

Default Cache at Installation Time

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

The following pages describe the way this single data cache is used. "Configuring the Data Cache to Improve Performance" on page 32-12 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 17-30 and "number of oam trips" on page 17-31 of the *System Administration Guide* for more information.
- Adaptive Server may choose to use the LRU cache replacement strategy that does not flush other pages out of the cache with pages that are used only once for an entire query.
- The checkpoint process ensures that if Adaptive Server needs to be restarted, the recovery process can be completed in a reasonable period of time. When the checkpoint process estimates that the number of changes to a database will take longer to recover than the configured value of the recovery interval configuration parameter, it traverses the cache, writing dirty pages to disk.
- The housekeeper task writes dirty pages to disk when idle time is available between user processes.

Effect of Data Cache on Retrievals

Figure 32-6 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.

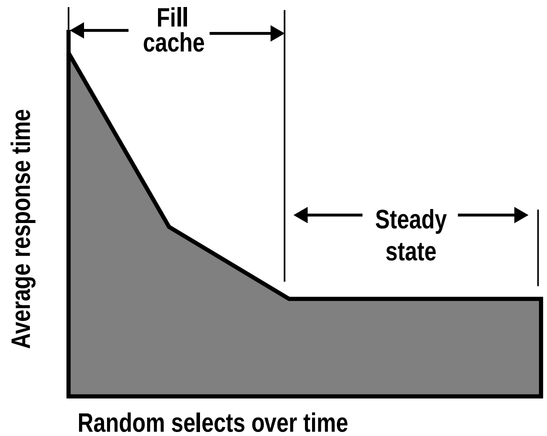


Figure 32-6: Effects of random selects on the data cache

If the cache is smaller than the total number of pages that are being accessed in all databases, there is a chance that a given statement will have to perform some disk I/O. A cache does not reduce the maximum possible response time—some query may still need to perform physical I/O for all of the pages it 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 32-7 shows how update transactions affect the average response time.

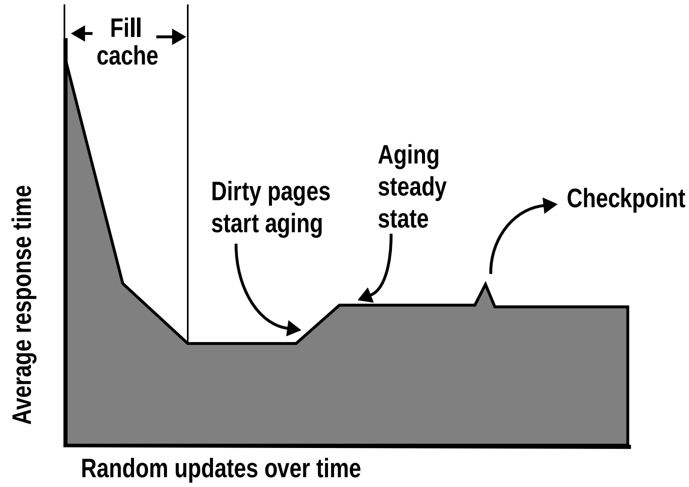


Figure 32-7: Effects of random data modifications on the data cache

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 these cases, if the next test steps request those pages, then no disk I/O will be needed.

Such a situation can bias the results away from a purely random test and lead to inaccurate performance estimates. The best testing strategy is to start with an empty cache or to make sure that all test steps access random parts of the database. For more precise testing, execute a mix of queries that is consistent with the planned mix of user queries on your system.

Cache Hit Ratio for a Single Query

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

To compute the cache hit ratio, use this formula:

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

With **statistics io**, physical reads are reported in I/O-size units. If a query uses 16K I/O, it reads 8 pages with each I/O operation. If **statistics io** reports 50 physical reads, it has read 400 pages. Use **showplan** to see the I/O size used by a query.

Cache Hit Ratio Information from *sp_sysmon*

sp_sysmon reports on cache hits and misses for:

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

The server-wide report provides the total number of cache searches and the percentage of cache hits and cache misses. See “Cache Statistics Summary (All Caches)” on page 39-75.

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 39-80.

Configuring the Data Cache to Improve Performance

When you install Adaptive Server, it has

- A single default data cache.
- The cache has only a 2K memory pool.
- The cache has a single spinlock (one cache partition).

To improve performance, you can split this cache into multiple named data caches and bind databases or database objects to them.

You can configure 4K, 8K, and 16K buffer 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.

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

Configuring the data cache can improve performance in the following ways:

- You can configure named data caches large enough to hold critical tables and indexes. This keeps other server activity from contending for cache space and speeds up queries using these tables, since the needed pages are always found in cache. 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 named data cache large enough to hold the “hot pages” of a table where a high percentage of the queries reference only a portion of the table. For example, if a table contains data for a year, but 75% of the queries reference data from the most recent month (about 8% of the table), configuring a cache of about 10% of the table size provides room to keep the most frequently used pages in cache and leaves some space for the less frequently used pages.
- You can assign tables or databases used in decision support (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.
- When changes are made to a cache by a user process, a **spinlock** denies all other processes access to the cache. Although spinlocks are held for extremely brief durations, they can slow performance in multiprocessor systems with high transaction rates. When you configure multiple caches, each cache is controlled by a separate spinlock, increasing concurrency on systems with multiple CPUs. Within a single cache, adding cache partitions creates multiple spinlocks to reduce contention. Spinlock contention is not an issue on single-engine servers.

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

Commands to Configure Named Data Caches

The commands used to configure caches and pools are shown in Table 32-1.

Table 32-1: Commands used to configure caches

Command	Function
sp_cacheconfig	Creates or drops named caches and set the size, cache type, cache policy and local cache partition number. Reports on sizes of caches and pools.
sp_poolconfig	Creates and drops I/O pools and changes their size, wash size, and asynchronous prefetch limit.
sp_bindcache	Binds databases or database objects to a cache.
sp_unbindcache	Unbinds the specified database or database object from a cache.
sp_unbindcache_all	Unbinds all databases and objects bound to a specified cache.

Table 32-1: Commands used to configure caches (continued)

Command	Function
sp_helpcache	Reports summary information about data caches and lists the databases and database objects that are bound to a cache. Also reports on the amount of overhead required by a cache.
sp_sysmon	Reports statistics useful for tuning cache configuration, including cache spinlock contention, cache utilization, and disk I/O patterns.

For a full description of configuring named caches and binding objects to caches, see Chapter 15, "Configuring Data Caches," in the *System Administration Guide*. Only a System Administrator can configure named caches and bind database objects to them.

Tuning Named Caches

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

- A cache that is poorly used hurts performance. If you allocate 25% of your data cache to a database that services a very small percentage of the query activity on your server, I/O increases in other caches.
- A pool that is unused hurts performance. If you add a 16K pool, but none of your queries use it, you have taken space away from the 2K pool. The 2K pool's cache hit ratio is reduced, and I/O is increased.
- A pool that is overused hurts performance. If you configure a small 16K pool, and virtually all of your queries use it, I/O rates are increased. The 2K cache will be under-used, while pages are rapidly cycled through the 16K pool. The cache hit ratio in the 16K pool will be very poor.
- When you balance your pool utilization within a cache, performance can increase dramatically. Both 16K and 2K queries experience improved cache hit ratios. The large number of pages often used by queries that perform 16K I/O do not flush 2K pages from disk. Queries using 16K will perform approximately one-eighth the number of I/Os required by 2K I/O.

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

Cache Configuration Goals

Goals for configuring caches are:

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

In addition to commands such as **showplan** and **statistics 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.

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 14, "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.

Deciding how to split the data 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 configuring caches is to reduce spinlock contention, increasing the number of cache partitions for heavily-used caches may be the only step. Moving a few high-I/O objects to separate caches also reduces the spinlock contention and improves performance.
- If your major goal is to improve response time by improving cache hit ratios for particular queries or applications, creating caches for the tables and indexes used by those queries should be guided by a thorough understanding of the access methods and I/O requirements.

Evaluating Cache Needs

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

If your databases and their logs are on separate logical devices, you can estimate cache proportions using **sp_sysmon** or operating system commands to examine physical I/O by device. See "Disk I/O Management" on page 39-92 for information about the **sp_sysmon** output showing disk I/O.

Large I/O and Performance

You can configure the default cache and any named caches you create for large I/O by splitting a cache into pools. The default I/O size is 2K, one Adaptive Server data page. For queries where pages are stored and accessed sequentially, Adaptive Server reads up to eight data pages in a single I/O. Since the majority of I/O time is spent doing physical positioning and seeking on the disk, large I/O can greatly reduce disk access time. In most cases, you want to configure a 16K pool in the default data cache.

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

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

- Queries that scan entire tables:

```
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
```

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

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

- 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.

- Queries such as **select into** that allocate large numbers of pages
- **create index** commands
- Bulk copy operations on heaps—both copy in and copy out
- The **update statistics**, **dbcc checktable**, and **dbcc checkdb** commands

The Optimizer and Cache Choices

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

The optimizer's knowledge is limited to the single query it is analyzing and to statistics about the table and cache. It does not have information about how many other queries are simultaneously using the same data cache, 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; often the best log I/O size is 4K.

Reducing Spinlock Contention with Cache Partitions

As the number of engines and tasks running on an SMP system increases, contention for the spinlock on the data cache can also increase. Any time a task needs to access the cache to find a page in cache or to relink a page on the LRU/MRU chain, it holds the cache spinlock to prevent other tasks from modifying the cache at the same time. With multiple engines and users, tasks wind up waiting for access to the cache. Adding cache partitions separates the cache into partitions that are each protected by its own spinlock. When a page needs to be read into cache or located, a hash function is applied to the database ID and page ID to identify which partition holds the page.

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

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

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

For more information on configuring cache partitions, see “Adding Cache Partitions” on page 15-29 of the *System Administration Guide*. For information on monitoring cache spinlock contention with **sp_sysmon**, see “Cache Spinlock Contention” on page 39-80.

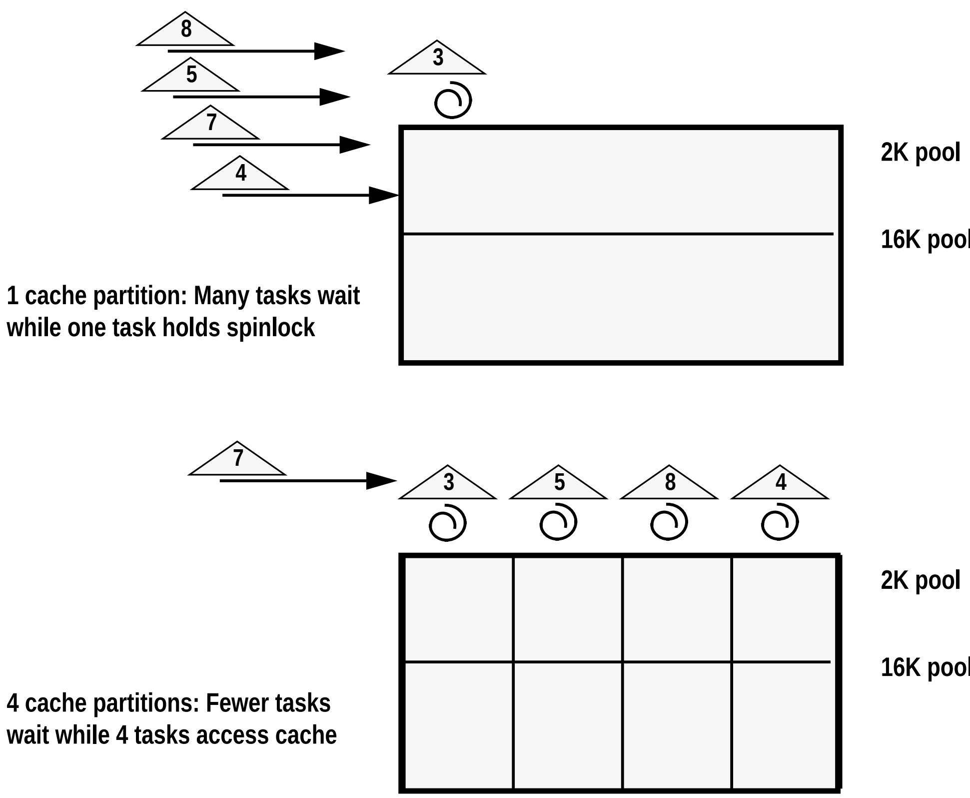


Figure 32-8: Cache partitions and spinlocks

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

Cache Replacement Strategies and Policies

The Adaptive Server optimizer uses two cache replacement strategies 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 3-14 and Figure 3-15 on page 3-22 for illustrations of these strategies. See “Specifying the Cache Strategy” on page 20-10 and “Controlling Large I/O and Cache Strategies” on page 20-12 for information on specifying the cache strategy in queries or setting values for tables.

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 LRU / MRU order.

The default cache replacement policy is strict replacement. Relaxed replacement policy should be used only when both of these conditions are true:

- There is little or no replacement of buffers in the cache
- The data is not updated or is updated infrequently

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

If you have created a cache to hold all, or most of, certain objects, and the cache hit rate is above 95%, using relaxed cache replacement policy for the cache can improve performance slightly. See "Configuring Cache Replacement Policy" on page 15-10 of the *System Administration Guide*.

Configuring Relaxed LRU Replacement for Database Logs

Log pages are filled with log records and are immediately written to disk. When applications include triggers, deferred updates or transaction rollbacks, some log pages may be read, but usually they are very recently used pages, which are still in the cache. Since accessing these pages in cache moves them to the MRU end of a strict-replacement policy cache, log caches may perform better with relaxed LRU replacement.

Relaxed LRU Replacement for Lookup Tables and Indexes

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

Named Data Cache Recommendations

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

- Create a named cache for *tempdb* and configure the cache for 16K I/O for use by **select into** queries and sorts.
- Create a named cache for the logs for your high-use databases. Configure pools in this cache to match the log I/O size set with **sp_logiosize**. See "Choosing the I/O Size for the Transaction Log" on page 32-25.
- If a table or its index is small and constantly in use, create a cache for just that object or for a few objects.
- For caches with cache hit ratios of more than 95%, configure relaxed LRU cache replacement policy if you are using multiple engines.
- Keep cache sizes and pool sizes proportional to the cache utilization objects and queries:
  - If 75% of the work on your server is performed in one database, that database should be allocated approximately 75% of the data cache, in a cache created specifically for the database, in caches created for its busiest tables and indexes, or in the default data cache.
  - If approximately 50% of the work in your database can use large I/O, configure about 50% of the cache in a 16K memory pool.
- It is better to view the cache as a shared resource than to try to micromanage the caching needs of every table and index. Start cache analysis and testing at the database level, concentrating on particular tables and objects with high I/O needs or high application priorities and those with special uses, such as *tempdb* and transaction logs.
- On SMP servers, use multiple caches to avoid contention for the cache spinlock:

- Use a separate cache for the transaction log for busy databases, and use separate caches for some of the tables and indexes that are accessed frequently.
- If spinlock contention is greater than 10% on a cache, split it into multiple caches or use cache partitions. Use **sp_sysmon** periodically during high-usage periods to check for cache contention. See "Cache Spinlock Contention" on page 39-80.
- Set relaxed LRU cache policy on caches with cache hit ratios of more than 95%, such as those configured to hold an entire table or index.

Sizing Caches for Special Objects, *tempdb*, and Transaction Logs

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

Determining Cache Sizes for Special Tables or Indexes

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

```
sp_spaceused table_name, 1
```

Examining Cache Needs for *tempdb*

Look at your use of *tempdb*:

- Estimate the size of the temporary tables and worktables generated by your queries. Look at the number of pages generated by **select into** queries. These queries can use 16K I/O, so you can use this information to help you size a 16K pool for the *tempdb* cache.
- Estimate the duration (in wall-clock time) of the temporary tables and worktables.
- Estimate how often queries that create temporary tables and worktables are executed. Try to estimate the number of simultaneous users, especially for queries that generate very large result sets in *tempdb*.

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

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

Examining Cache Needs for Transaction Logs

On SMP systems with high transaction rates, binding the transaction log to its own cache can greatly reduce cache spinlock contention in the default data cache. In many cases, the log cache can be very small.

The current page of the transaction log is written to disk when transactions commit, so your objective in sizing the cache or pool for the transaction log is not to avoid writes. Instead, you should try to size the log to reduce the number of times that processes that need to reread log pages must go to disk because the pages have been flushed from the cache. Adaptive Server processes that need to read log pages are:

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

When sizing a cache for a transaction log:

- Examine the duration of processes that need to reread log pages. Estimate how long the longest triggers and deferred updates last. If some of your long-running transactions are rolled back, check the length of time they ran.
- Estimate the rate of growth of the log during this time period. You can check your transaction log size with **sp_spaceused** at regular intervals to estimate how fast the log grows.

Use this log growth estimate and the time estimate to size the log cache. For example, if the longest deferred update takes 5 minutes, and the transaction log for the database grows at 125 pages per minute, 625 pages are allocated for the log while this transaction executes. If a few transactions or queries are especially long-running, you may want to size the log for the average, rather than the maximum, length of time.

Choosing the I/O Size for the Transaction Log

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

- When a transaction ends
- When the user log cache is full
- When the transaction changes tables in another database
- When another process needs to write a page referenced in the user log cache
- At certain system events

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

In environments with high transaction rates or transactions that create large log records, the 2K transaction log pages fill quickly, and a large proportion of log writes are due to full log pages, rather than group commits. Creating a 4K pool for the transaction log can greatly reduce the number of log writes in these environments.

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

- Your database is using 2K log I/O
- The number of log writes per second is high
- The average number of writes per log page is slightly above one

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

```
                         per sec   per xact    count  % of total
Transaction Log Writes      22.5      458.0     1374     n/a
Transaction Log Alloc       20.8      423.0     1269     n/a
Avg # Writes per Log Page    n/a        n/a  1.08274     n/a
```

See "Transaction Log Writes" on page 39-51 for more information.

Configuring for Large Log I/O Size

The log I/O size for each database in reported the server's error log when Adaptive Server starts. You can also use **sp_logiosize**. To see the size for the current database, execute **sp_logiosize** with no parameters. To see the size for all databases on the server and the cache in use by the log, use:

```
sp_logiosize "all"
```

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

```
sp_logiosize "default"
```

By default, Adaptive Server sets the log I/O size for user databases to 4K. If no 4K pool is available in the cache used by the log, 2K I/O is used instead.

If a database is bound to a cache, all objects not explicitly bound to other caches use the database's cache. This includes the *syslogs* table. To bind *syslogs* to another cache, you must first put the database in single-user mode, with **sp_dboption**, and then use the database and execute **sp_bindcache**. Here is an example:

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

Additional Tuning Tips for Log Caches

For further tuning after configuring a cache for the log, check **sp_sysmon** output. Look at the output for:

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

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

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

Basing Data Pool Sizes on Query Plans and I/O

Divide a cache into pools based on the proportion of the I/O performed by your queries that use the corresponding I/O sizes. If most of your queries can benefit from 16K I/O, and you configure a very small 16K cache, you may see worse performance. Most of the space in the 2K pool 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 nested-loop join queries that have to repeatedly reread the inner table from disk.

Making a good choice about pool sizes requires:

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

Checking I/O Size for Queries

You can examine query plans and I/O statistics to determine which queries are likely to perform large I/O and the amount of I/O those queries perform. This information can form the basis for estimating the amount of 16K I/O the queries should perform with a 16K memory pool. 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 "Large I/O and Performance" on page 32-17 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 most I/O occurs in point queries using indexes to access a small number of rows, make the 16K pool relatively small, say about 10 to 20% of the cache size.
- If you estimate that a large percentage of the I/Os will use the 16K pool, configure 50 to 75% of the cache for 16K I/O. Queries that use 16K I/O include any query that 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% of your cache space in a 16K pool,

and use **showplan** and **statistics i/o** while you run your queries. Examine the **showplan** output for the "Using 16K I/O" message. Check **statistics i/o** output to see how much I/O is performed.

- If you think that your typical application mix uses both 16K I/O and 2K I/O simultaneously, configure 30 to 40% of your cache space for 16K I/O. Your optimum may be higher or lower, depending on the actual mix and the I/O sizes chosen by the query. If many tables are accessed by both 2K I/O and 16K I/O, Adaptive Server cannot use 16K I/O, if any page from the extent is in the 2K cache. It performs 2K I/O on the other pages in the extent that are needed by the query. This adds to the I/O in the 2K cache.

After configuring for 16K I/O, check cache usage and monitor the I/O for the affected devices, using **sp_sysmon** or Adaptive Server Monitor. Also, use **showplan** and **statistics io** to observe your queries.

- Look for nested-loop join queries where an inner table would use 16K I/O, and the table is repeatedly scanned using 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 20-9.

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 15-20 of the *System Administration Guide* for more information.

Overhead of Pool Configuration and Binding Objects

Configuring memory pools and binding objects to caches can affect users on a production system, so these activities are best performed during off-hours.

Pool Configuration Overhead

When a pool is created, deleted, or changed, the plans of all stored procedures and triggers that use objects bound to the cache are recompiled the next time they are run. If a database is bound to the cache, this affects all of the objects in a database.

There is a slight amount of overhead involved in moving buffers between pools.

Cache Binding Overhead

When you bind or unbind an object, all the object's pages that are currently in the cache are flushed to disk (if dirty) or dropped from the cache (if clean) during the binding process. The next time the pages are needed by user queries, they must be read from the disk again, slowing the performance of the queries.

Adaptive Server acquires an exclusive lock on the table or index while the cache is being cleared, so binding can slow access to the object by other users. The binding process may have to wait until 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. **optdiag** reports a statistics called “Large I/O efficiency” for tables and indexes. When this value is 1, or close to 1, large I/O is very efficient. As the value drops, more I/O is required to access data pages needed for a query, and large I/O may be bringing pages into cache that are not needed by the query. You should consider rebuilding indexes when large I/O efficiency drops or activity in the pool increases due to increased 16K I/O.

When large I/O efficiency drops, you can:

- Run **reorg rebuild** on tables that use data-only-locking. You can also use **reorg rebuild** on the index of data-only-locked tables.
- For allpages-locked tables, drop and re-create the indexes.

For more information, see “Running reorg on Tables and Indexes” on page 30-1.

Diagnosing Excessive I/O Counts

There are several reasons why a query that performs large I/O might require more reads than you anticipate:

- The cache used by the query has a 2K cache and other processes have brought pages from the table into the 2K cache. If Adaptive Server finds that one of the pages it would read using 16K I/O 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 and clustered indexes on data-only-locked tables, an extent may store both leaf-level pages and pages from higher levels of the index. 2K I/O is performed on the higher levels of indexes, and for leaf-level pages when few pages are needed by a query. When a covering leaf-level scan performs 16K I/O, it is likely that some of the pages from some extents will

be in the 2K cache. The rest of the pages in the extent will be read using 2K I/O.

Using *sp_sysmon* to Check Large I/O Performance

The **sp_sysmon** output for each data cache includes information that can help you determine the effectiveness for large I/Os:

- "Large I/O Usage" on page 39-85 reports the number of large I/Os performed and denied and provides summary statistics.
- "Large I/O Detail" on page 39-86 reports the total number of pages that were read into the cache by a large I/O and the number of pages that were actually accessed while they were in the cache.

Speed of Recovery

As users modify data in Adaptive Server, only the transaction log is written to disk immediately, 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 these benefits:

- Many transactions may change a page in the cache or read the page in the cache, but only one physical write is performed.
- Adaptive Server performs many physical writes at times when the I/O does not cause contention with user processes.

Tuning the Recovery Interval

The default recovery interval in Adaptive Server is five minutes per database. Changing the recovery interval can affect performance because it can impact the number of times Adaptive Server writes pages to disk. Table 32-2 shows the effects of changing the recovery interval from its current setting on your system.

Table 32-2: Effects of recovery interval on performance and recovery time

Setting	Effects on Performance	Effects on Recovery
Lower	May cause more reads and writes and may lower throughput. Adaptive Server will write dirty pages to the disk more often. Any checkpoint I/O "spikes" will be smaller.	Recovery period will be very short.
Higher	Minimizes writes and improves system throughput. Checkpoint I/O spikes will be higher.	Automatic recovery may take more time on start-up. Adaptive Server may have to reapply a large number of transaction log records to the data pages.

See "recovery interval in minutes" on page 17-24 of the *System Administration Guide* for information on setting the recovery interval. **sp_sysmon** reports the number and duration of checkpoints. See "Recovery Management" on page 39-88.

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 39-88.

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 32-9 on page 32-34.

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 32-9. 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.

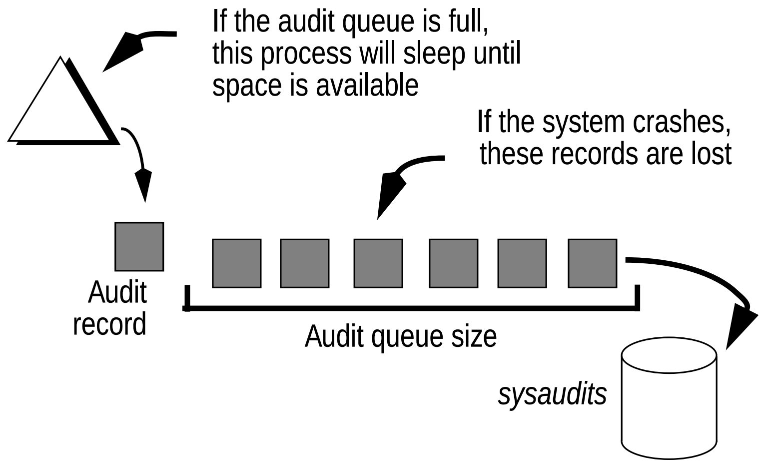


Figure 32-9: Trade-offs in auditing and performance

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.

33 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

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 39, "Monitoring Performance with sp_sysmon." Pay special attention to the output associated with the discussions in:

- "I/O Device Contention" on page 39-29
- "APL Heap Tables" on page 39-42
- "Last Page Locks on Heaps" on page 39-67
- "Disk I/O Management" on page 39-92

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 33-1 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.

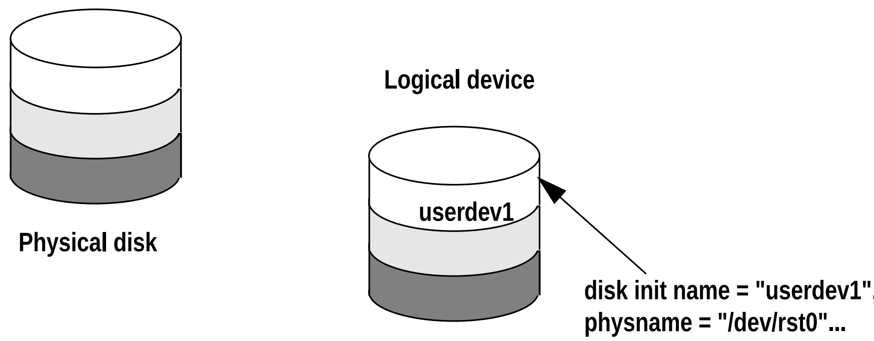


Figure 33-1: Physical and logical disks

- 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 block of storage for a table. Partitioning a table splits it so that multiple tasks can access it simultaneously. When partitioned tables are placed on segments with a matching number of devices, each partition starts on a separate database device.

Use **sp_helpdevice** to get information about devices, **sp_helpsegment** to get information about segments, and **sp_helpartition** 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 33-2 shows desirable and undesirable data distribution.

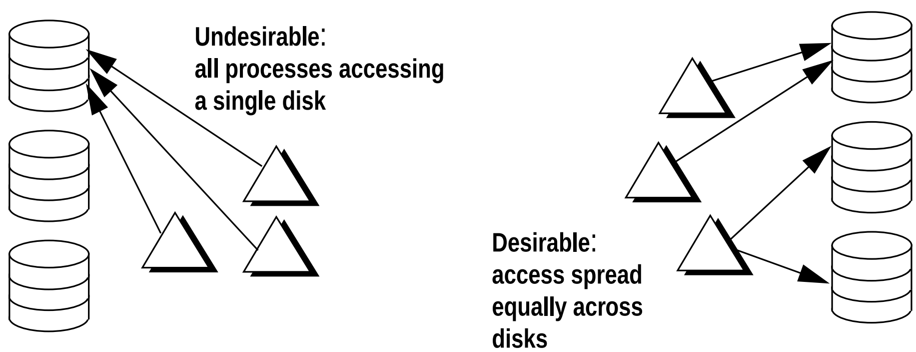


Figure 33-2: Spreading I/O across disks

Avoiding Physical Contention in Parallel Join Queries

The example in Figure 33-3 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.

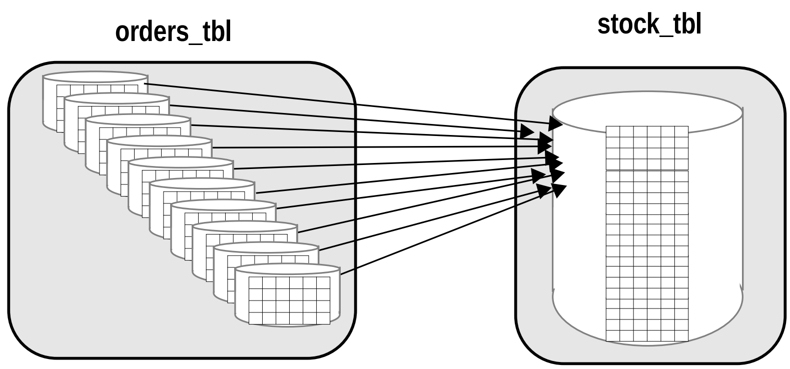


Figure 33-3: Joining tables on different physical devices

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 33-4.

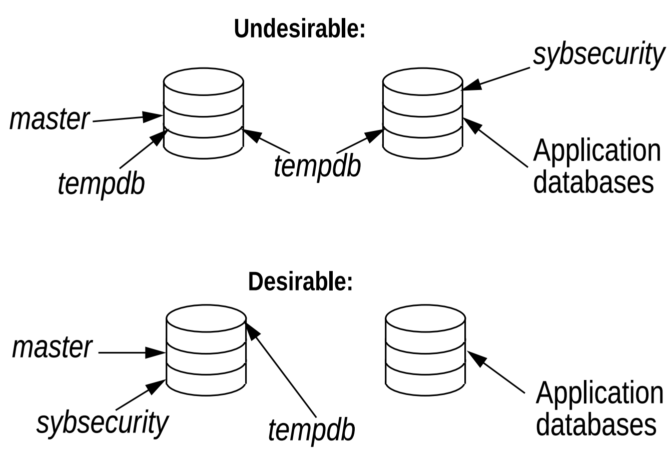


Figure 33-4: Isolating database I/O from server-wide I/O

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.

See Chapter 35, “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 causes such a dangerous reliability problem that both **create database** and **alter database** require the use of the **with override** option to put the transaction log on the same device as the data itself.

Placing the log on a separate segment limits log size, which keeps it from competing with other objects for disk space. Placing the log on a separate physical disk:

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

- Speeds recovery, since simultaneous asynchronous prefetch requests can read ahead on both the log device and the data device without contention

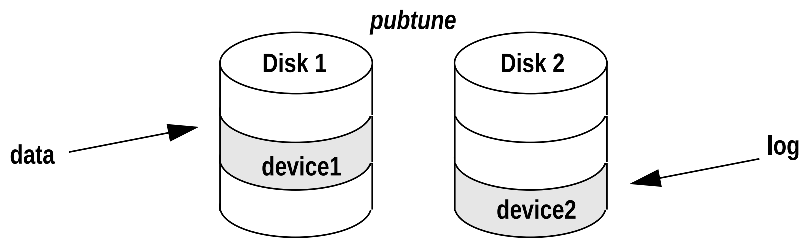


Figure 33-5: Placing log and data on separate physical disks

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.

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 21-9 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 33-6. Disk hardware failure often results in whole physical disks being lost or unavailable.

Mirroring on separate disks also minimizes the performance impact of mirroring.

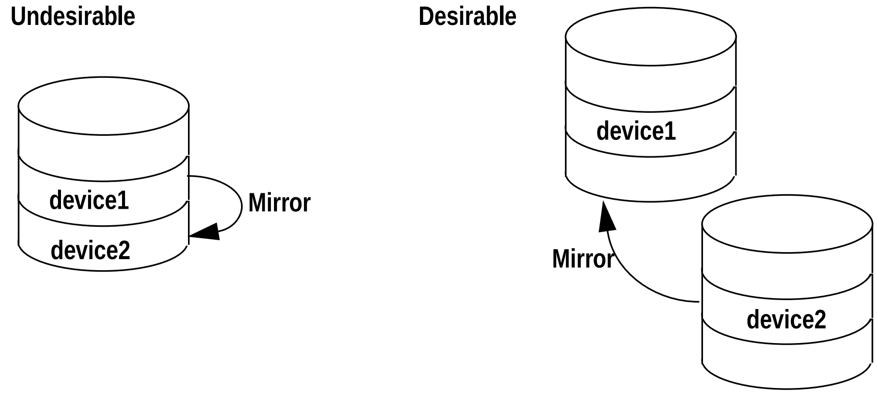


Figure 33-6: Mirroring data to separate physical disks

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 13, "Mirroring Database Devices," in the *System Administration Guide* for more information on mirroring.

Adaptive Server provided disk mirroring for database devices before disk mirroring was available on many operating systems. Disk mirroring is now widely available at the operating system level, and performance of OS-level mirroring is generally better than using Adaptive Server mirroring. If you do not use mirroring, or use operating system mirroring, set the configuration parameter **disable disk mirroring** to 1. This may yield slight performance improvements.

Mirroring 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 33-7. Reads always come from the primary side. Disk mirroring has no effect on the time required to read data.

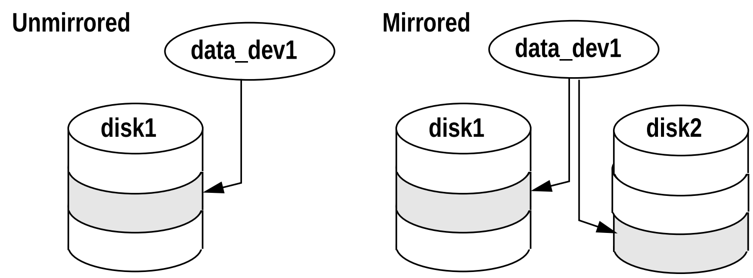


Figure 33-7: Impact of mirroring on write performance

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 a label that points to one or more database devices. Figure 33-8 shows the segment named *segment1*; this segment includes three database devices, *data_dev1*, *data_dev2*, and *data_dev3*. Each database 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.

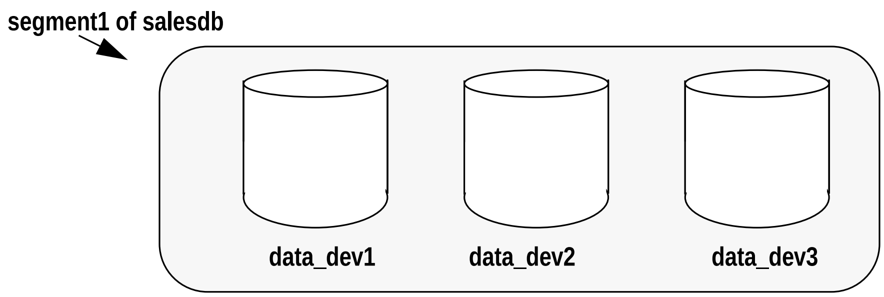


Figure 33-8: Segment labeling a set of disks

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. You cannot place a clustered index on a separate segment than its data pages. When you create a clustered index, using the **on *segment_name*** clause, the entire table is moved to the specified segment, and the clustered index tree is built there.

You can improve performance by placing nonclustered indexes on a separate segment, as shown in Figure 33-9.

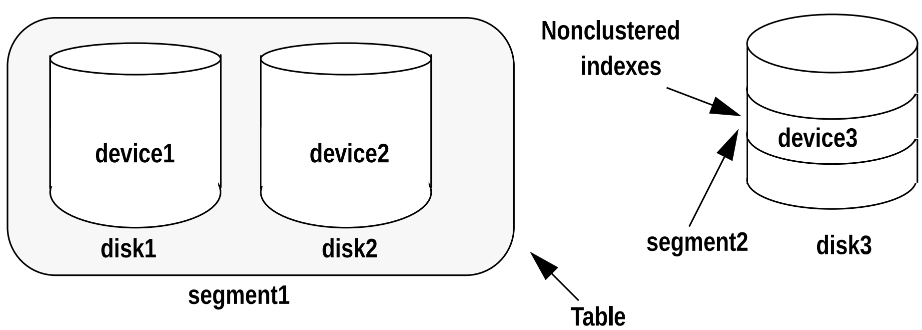


Figure 33-9: Separating a table and its nonclustered indexes

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 33-10. 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.

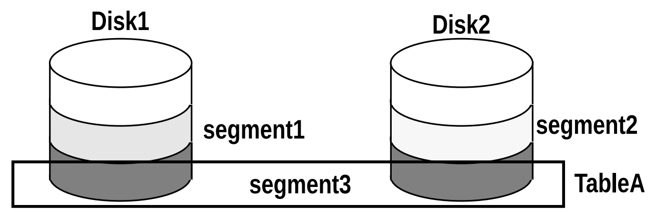


Figure 33-10: Splitting a large table across devices with segments

See "Splitting Tables" on page 23-5 in the *System Administration Guide* for more information.

Moving Text Storage to a Separate Device

When a table includes a *text*, *image*, or Java off-row datatype, the table itself stores a pointer to the data value. The actual data is stored on a separate linked list of pages called a LOB (large object) chain. Writing or reading a LOB value requires at least two disk accesses, one to read or write the pointer and one for subsequent reads or writes for the data. If your application frequently reads or writes these values, you can improve performance by placing the LOB chain on a separate physical device, as shown in Figure 33-11. Isolate LOB chains on disks that are not busy with other application-related table or index access.

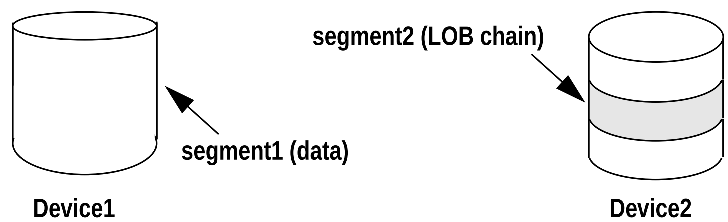


Figure 33-11: Placing the text chain on a separate segment

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

You can use **sp_placeobject** to move all future allocations for the LOB columns to a separate segment. See "Placing Text Pages on a Separate Device" on page 23-14 of the *System Administration Guide* for more information.

Partitioning Tables for Performance

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

- Partitioning allows parallel query processing to access each partition of the table. Each worker process in a partitioned-based scan reads a separate partition.
- Partitioning makes it possible to load a table in parallel with bulk copy. For more information on parallel **bcp**, see the *Utility Programs* manual.
- Partitioning makes it possible to distribute a table's I/O over multiple database devices.
- Partitioning provides multiple insertion points for a heap table.

The tables you choose to partition depend on the performance issues you 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 11-20 for more information and examples of partitioning to meet specific performance goals.

User Transparency

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

- If queries do not include **order by** or other commands that require a sort, data returned by a parallel query may not in the same order as data returned by serial queries.

- The **dbcc checktable** and **dbcc checkdb** commands list the number of data pages in each partition. See Chapter 25, "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 simultaneous access by worker processes. When enough worker processes are available, and the value for the **max parallel degree** configuration parameter is set equal to or greater than the number of partitions, one worker process scans each of the table's partitions.

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

The optimizer can choose to use parallel query processing for a query against a partitioned table when parallel query processing is enabled. The optimizer considers a parallel partition scan for a query when the base table for the query is partitioned, and it considers a parallel index scan for a useful index. See Chapter 12, "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 an allpages-locked heap table attempt to insert the rows on the last page of the table. If multiple users insert data simultaneously, each new insert transaction must wait for the previous transaction to complete in order to proceed. Partitioning an allpages-locked heap table improves the performance of concurrent inserts by reducing contention for the last page of a page chain.

For data-only-locked tables, Adaptive Server stores one or more hints that point to a page where an insert was recently performed. Blocking during inserts on data-only-locked tables occurs only with high rates of inserts. Partitioning data-only-locked heap tables increases the number of hints, and can help if inserts are blocking.

How Partitions Address Page Contention

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

Figure 33-12 shows an example of insert activity in an allpages-locked heap table with three partitions.

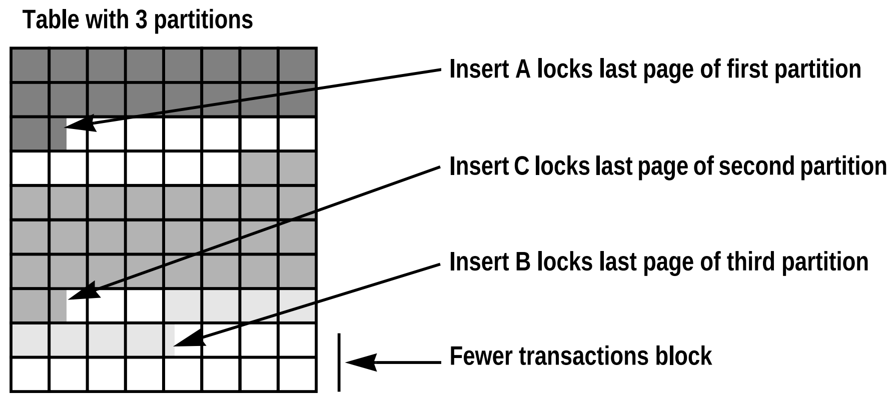


Figure 33-12: Addressing allpages-locked heap contention with partitions

Selecting Heap Tables to Partition

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

- Use **sp_sysmon** to look for last page locks on heap tables. See "Lock Management" on page 39-62.
- Use **sp_object_stats** to report on lock contention. See "Identifying Tables Where Concurrency Is a Problem" on page 28-11.

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 33-22 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 17, “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 33-1 illustrates how Adaptive Server assigns 5 partitions to 3, 5, and 12 devices, respectively.

Table 33-1: Assigning partitions to segments

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

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

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

RAID Devices and Partitioned Tables

Table 33-1 and other statements in this chapter describe the Adaptive Server logical devices that map to a single physical device. A striped RAID device may contain multiple physical disks, but it appears to Adaptive Server as a single logical device. For a striped RAID device, you can use multiple partitions on the single logical device and achieve good parallel query performance.

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

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

Space Planning for Partitioned Tables

When planning for partitioned tables, the two major issues are:

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

How you make these decisions depends on:

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

You need to estimate how often your partitioned tables need maintenance: some applications need frequent index re-creation to maintain balance, while others need little maintenance. For those applications that need frequent load balancing for performance, having space to re-create a clustered index or run **reorg rebuild** provides the speediest and easiest method. However, since creating clustered indexes requires copying the data pages, the space available on the segment must be equal to approximately 120% of the space occupied by the table. See “Determining the Space Available for Maintenance Activities” on page 30-10 for more information.

The following descriptions of read-only, read-mostly, and random data modification provide a general picture of the issues involved in object placement and in maintaining partitioned tables. See “Steps for Partitioning Tables” on page 33-32 for more information about the specific tasks required during maintenance.

Read-Only Tables

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

If a clustered index is needed, the table's data pages can occupy up to 80% of the space in the segment. The clustered index tree requires about 20% of the space used by the table. This size varies, depending on the length of the key. Loading the data into the table initially and creating the clustered index requires several steps, but once you have performed these steps, maintenance is minimal.

Read-Mostly Tables

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

- If there are inserts to the table, and the clustered index key does not balance new space allocations evenly across the partitions, the disks underlying some partitions may become full and new extent allocations will be made to a different physical disk, 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_helpartition** to check for partitions that have disproportionate numbers of pages. If the imbalance in partition size leads to degradation in parallel query response times or optimization, you may want to balance the distribution by using one of the methods described in "Steps for Partitioning Tables" on page 33-32.
- 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. "Using bcp to Correct Partition Balance" on page 33-27.

Tables with Random Data Modification

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

- More pages must be read to access a given number of rows, requiring additional I/O and wasting data cache space.
- On tables that use allpages locking, the performance of large I/O and asynchronous prefetch will almost certainly suffer because the page chain crosses extents and allocation units. Buffers brought in by large I/O may be flushed from cache before all of the pages are read. The asynchronous prefetch look-ahead set size is reduced by cross-allocation unit hops while following the page chain.

Once the fragmentation starts to take its toll on application performance, you need to perform maintenance. If that requires dropping and re-creating the clustered index, you need 120% of the space occupied by the table. If 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 33-32.

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.

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

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

```
alter table historytab partition 10
```

alter table...unpartition Syntax

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

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

alter table *table_name* unpartition

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

```
alter table historytab unpartition
```

Changing the Number of Partitions

To change the number of partitions in a table, first unpartition the table using **alter table...unpartition** (see "alter table...unpartition Syntax" on page 33-22). Then use **alter table...partition**, specifying the new number of partitions. This does not move the existing data in the table.

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

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

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

Distributing Data Evenly Across Partitions

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

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

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 11-11. See Chapter 13, "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 reorg rebuild on Data-Only-Locked Tables

The **reorg rebuild** command copies data rows in data-only-locked tables to new data pages. If there is a clustered index, rows are copied in clustered key order. Running **reorg rebuild** redistributes data evenly on partitions. The clustered index and any nonclustered indexes are rebuilt. To run **reorg rebuild** on the table, provide only the table name:

```
reorg rebuild titles
```

Using *drop index* and *create clustered index*

If the index on the table was created with **create index**, 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 30-3 for options.

Using Parallel *bcp* to Copy Data into Partitions

Loading data into a partitioned table using parallel **bcp** lets you direct the data to a particular partition in the table. Before you run parallel bulk copy, the table should be located on the segment, and it should be partitioned. You should drop all indexes, so that you do not experience failures due to index deadlocks. Use **alter table...disable trigger** so that fast, minimally-logged bulk copy is used, instead of slow bulk copy, which is completely logged. You may also want to set the database option **trunc log on chkpt** to keep the log from filling up during large loads.

You can use operating system commands to split the file into separate files, and then copy each file, or use the **-F** (first row) and **-L** (last row) command line flags for **bcp**. Whichever method you choose, be sure that the number of rows sent to each partition is approximately the same.

Here is an example using separate files:

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

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

```
bcp mydb..huge_tab:1 in bigfile -F1 -L100000
bcp mydb..huge_tab:2 in bigfile -F100001 -L200000
...
bcp mydb..huge_tab:10 in bigfile -F900001 -L1000000
```

If you have space to split the file into multiple files, copying from separate files is much faster than using the first row and last row command-line arguments, since **bcp** needs to parse each line of the input file when using **-F** and **-L**. This parsing process can be very slow, almost negating the benefits from parallel copying.

Parallel Copy and Locks

Starting many current parallel **bcp** sessions may cause Adaptive Server to run out of locks. When you copy in to a table, **bcp** acquires an exclusive intent lock on the table, and either page or row locks, depending on the locking scheme. If you are copying in very large tables, and especially if you are performing simultaneous copies into a partitioned table, this can require a very large number of locks. To avoid running out of locks:

- Set the **number of locks** configuration parameter high enough, or
- Use the **-b** *batchsize* **bcp** flag to copy smaller batches. If you do not use the **-b** flag, the entire copy operations is treated as a single batch.

For more information on **bcp**, see the *Utility Programs* manual.

Getting Information About Partitions

sp_helpartition prints information about table partitions. For partitioned tables, it shows the number of data pages in the partition and summary information about data distribution. Issue **sp_helpartition**, giving the table name. This example shows data distribution immediately after creating a clustered index:

```
sp_helpartition sales
```

```
partitionid firstpage   controlpage ptn_data_pages
 ----------- ----------- ----------- --------------
           1        6601        6600           2782
           2       13673       13672           2588
           3       21465       21464           2754
           4       29153       29152           2746
           5       36737       36736           2705
           6       44425       44424           2732
           7       52097       52096           2708
           8       59865       59864           2755
           9       67721       67720           2851

(9 rows affected)
 Partitions  Average Pages Maximum Pages Minimum Pages Ratio (Max/Avg)
 ----------- ------------- ------------- ------------- -----------------
           9          2735          2851          2588          1.042413
```

sp_helpartition shows how evenly data is distributed between partitions. The final column in the last row shows the ratio of the average column size to the maximum column size. This ratio is used to determine whether a query can be run in parallel. If the maximum is twice as large as the average, the optimizer does not choose a parallel plan. Uneven distribution of data across partitions is called **partition skew**.

If a table is not partitioned, **sp_helpartition** prints the message “Object is not partitioned.” When used without a table name, **sp_helpartition** prints the names of all user tables in the database and the number of partitions for each table. **sp_help** calls **sp_helpartition** when used with a table name.

Using *bcp* to Correct Partition Balance

If you need to load additional data into a partitioned table that does not have clustered indexes, and **sp_helpartition** shows that some partitions contain many more pages than others, you can use the bulk copy session to help balance number of rows on each partition.

The following example shows that the table has only 487 pages on one partition, and 917 on another:

```
partitionid firstpage   controlpage ptn_data_pages
----------- ----------- ----------- --------------
          1      189825      189824            812
          2      204601      204600            487
          3      189689      189688            917

(3 rows affected)
Partitions Average Pages Maximum Pages Minimum Pages Ratio (Max/Avg)
---------- ------------- ------------- ------------- ---------------
         3           738           917           487        1.242547
```

The number of rows to add to each partition can be computed by:

- Determining the average number of rows that would be in each partition if they were evenly balanced, that is, the sum of the current rows and the rows to be added, divided by the number of partitions.
- Estimating the current number of rows on each partition, and subtracting that from the target average

The formula can be summarized as:

```
Rows to add = (total_old_rows + total_new_rows)/#_of_partitions
      - rows_in_this_partition
```

This sample procedure uses values stored in *systabstats* and *syspartitions* to perform the calculations:

```
create procedure help_skew @object_name varchar(30), @newrows int
as
declare @rows int, @pages int, @rowsperpage int,
           @num_parts int
select @rows = rowcnt, @pages = pagecnt
      from systabstats
      where id = object_id(@object_name) and indid in (0,1)
select @rowsperpage = floor(@rows/@pages)
select @num_parts = count(*) from syspartitions
      where id = object_id(@object_name)

select partitionid,  (@rows + @newrows)/@num_parts -
      ptn_data_pgs(id, partitionid)*@rowsperpage as rows_to_add
      from syspartitions
      where id = object_id (@object_name)
```

To use this procedure to determine how many rows to add to each partition in the *customer* table, when 18,000 rows need to be copied in, use:

```
help_skew customer, 18000
```

The results are:

```
 partitionid rows_to_add
----------- -----------
          1        5255
          2        9155
          3        3995
```

➤ ***Note***

If the partition skew is large, and the number of rows to be added is small, this procedure returns negative numbers for those rows that contain more than the average number of final rows. Also, the query results are more accurate if **update statistics** and **update partition statistics** have been run so that table and partition statistics are up to date.

With the results from **help_skew**, you can then split the file containing the data to be loaded into separate files of that length, or use the **-F** (first) and **-L** (last) flags to **bcp**. See "Using bcp to Correct Partition Balance" on page 33-27.

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_helpartition** 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_helpartition**. 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 33-42 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 25, "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 or running **reorg rebuild** automatically redistributes the data within partitions and updates the partition statistics. **dbcc checktable**, **dbcc checkdb**, and **dbcc checkalloc** also update partition statistics as they perform checks.

Syntax for *update partition statistics*

Its syntax is:

```
update partition statistics table_name
   [partition_number]
```

Use **sp_helpartition** 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 23, "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. This section provides examples for the following situations:

- The table has not been created and populated yet.
- The table exists, but it is not on the database segment where you want the table to reside.
- The table exists on the segment where you want it to reside, and you want to redistribute the data to improve performance, or you want to add devices to the segment.

➤ ***Note***

The following sections provide procedures for a number of situations, including those in which severe space limitations in the database make partitioning and creating clustered indexes very difficult. These complex procedures are needed only in special cases. If you have ample room on your database devices, the process of partitioning and maintaining partitioned table performance requires only a few simple steps.

Backing Up the Database After Partitioning Tables

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.

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 and load the data with **bcp** 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 33-11.
2. Partition the table, with one partition for each physical device in the segment. See “alter table...partition Syntax” on page 33-22.

➤ ***Note***

If the input data file is not in clustered key order, and the table will occupy more than 40% of the space on the segment, and you need a clustered index, see “Special Procedures for Difficult Situations” on page 33-39.

3. Copy the data into the table using parallel bulk copy. See “Using Parallel bcp to Copy Data into Partitions” on page 33-25 for examples using **bcp**.
4. If you do not need a clustered index, use **sp_helpartition** to verify that the data is distributed evenly on the partitions. See “Getting Information About Partitions” on page 33-26.

   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 simultaneously 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 33-24). To create a clustered index with referential integrity constraints, use **alter table...add constraint** (see “Using Constraints and alter table” on page 33-24).

   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% 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 33-36.

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 33-22.
2. Partition the table, matching the number of devices on the target segment. See “alter table...partition Syntax” on page 33-22.
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 33-24) or **alter table...drop constraint** (see “Using Constraints and alter table” on page 33-24.).
4. Create or re-create the clustered index with the **on** *segment_name* clause. When the segment name is different from the current segment where the table is stored, creating the clustered index performs a parallel sort and distributes the data evenly on the partitions as it copies the rows to match the index order. This step re-creates the nonclustered indexes on the table. See “Distributing Data Evenly Across Partitions” on page 33-23.
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% of the space in the segment, you can create a clustered index to redistribute the data.

If the table occupies more than 40 to 45% of the space on the segment, you need to bulk copy the data out, truncate the table, and copy the data in again. The steps you take depend on whether you need a clustered index and whether the data is already in clustered key order.

Use **sp_helpsegment** and **sp_spaceused** to see if there is room to create a clustered index on the segment.

If There Is Enough Space to Create or Re-create the Clustered Index

If there is enough space, see “Distributing Data Evenly Across Partitions” on page 33-23 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 there is space available on another segment:

1. Create a clustered index, specifying the segment where the space exists. This moves the table to that location.
2. Drop the index.
3. Re-create the clustered index, specifying the segment where you want the data to reside.
4. Dump the database.

If There Is Not Enough Space to Re-create the Clustered Index

If there is not enough space, and you need a to re-create a clustered index on the tables:

1. Copy out the data using bulk copy.
2. Unpartition the table. See "alter table...unpartition Syntax" on page 33-22.
3. Truncate the table with **truncate table**.
4. Drop the clustered index using **drop index** or **alter table...drop constraint**. See "Distributing Data Evenly Across Partitions" on page 33-23. Also, drop nonclustered indexes, to avoid deadlocking during the parallel bulk copy sessions.
5. Repartition the table. See "alter table...partition Syntax" on page 33-22.
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 33-25.
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 33-22.
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 33-22.
6. Copy the data in using parallel bulk copy. See “Using Parallel bcp to Copy Data into Partitions” on page 33-25.
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% 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 33-22.
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 33-22.
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 33-25.
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 33-42. 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 33-22.
5. Repartition the table, specifying the new number of devices in the segment. See “alter table...partition Syntax” on page 33-22.
6. If a clustered index exists, drop 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 33-23.
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% 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 33-22.
3. Truncate the table.
4. Repartition the table. See “alter table...partition Syntax” on page 33-22.
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 33-24) or **alter table...drop constraint** (see “Using Constraints and alter table” on page 33-24).
6. Use parallel bulk copy to copy in enough data to fill approximately 40% of the segment. This must be a representative sample of the values in the key column(s) of the clustered index. Copying in 40% of the data is much more likely to yield good results than smaller amounts of data, 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 33-25.
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. The clustered index directs the rows to the correct partitions.
9. Check the distribution of data pages on partitions with **sp_helpartition**, 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 can cause page splitting on the data pages, taking slightly more room in the database. However, once the clustered index exists, and all the data is loaded, future maintenance activities can use simpler and faster methods.

A Complex Alternative for Clustered Indexes

This set of steps may be useful when:

- The table data occupies more than 40 to 45% of the segment.
- The table data is not in clustered key order, and you need to create a clustered index.
- You do not get satisfactory results trying to load a representative sample of the data, as explained in "A Technique for Clustered Indexes on Large Tables" on page 33-39.

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 33-24 or "Using Constraints and alter table" on page 33-24.

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 record the partition numbers and values in the output. This example shows the values for a table on four partitions:

   ```
   create clustered index order_cix
       on orders(order_id)
   ```

```
The Create Index is done using Parallel Sort
Sort buffer size: 1500
Parallel degree: 25
Number of output devices: 3
Number of producer threads: 4
Number of consumer threads: 4
The distribution map contains 3 element(s) for 4
partitions.
Partition Element: 1

450977
Partition Element: 2

903269
Partition Element: 3

1356032
Number of sampled records: 2449
```

These values, together with the minimum value from step 1, are the key values that the sort uses as 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 33-22.
7. Truncate the table.
8. Repartition the table. See “alter table...partition Syntax” on page 33-22.
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 clustered index forces the pages to the correct partition.
13. Check the balance of data pages on the partitions with **sp_helpartition** 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. The entire table must be copied by a single, slow bulk copy. Also, the clustered index is likely to lead to page splitting on the data pages if the table uses allpages locking, so more space might be required.

Problems When Devices for Partitioned Tables Are Full

Simply adding disks and re-creating indexes when partitions are full may not solve load-balancing problems. If a physical device that underlies a partition becomes completely full, the data-copy stage of re-creating an index cannot copy data to that physical device. If a physical device is almost completely full, re-creating the clustered index does not always succeed in establishing a good load balance.

Adding Disks When Devices Are Full

The result of creating a clustered index when a physical device is completely full is that two partitions are created on one of the other physical devices. Figure 33-13 and Figure 33-14 show one such situation.

Devices 2 and 3 are completely full, as shown in Figure 33-13.

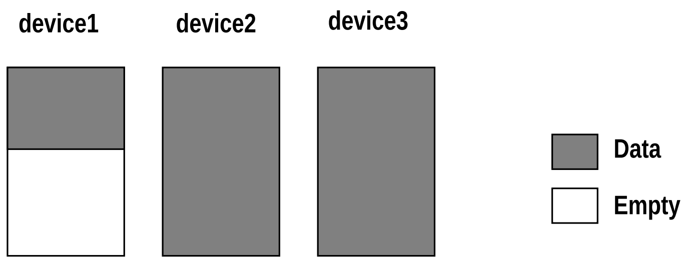


Figure 33-13: A table with 3 partitions on 3 devices

Adding two devices, repartitioning the table to use five partitions, and dropping and re-creating the clustered index produces the following results:

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

Figure 33-14 shows these results.

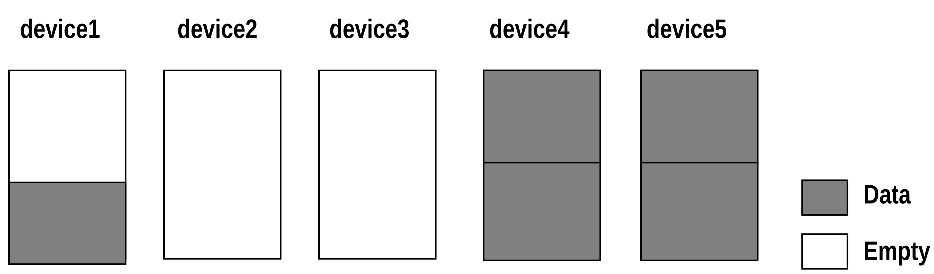


Figure 33-14: Devices and partitions after create index

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

Adding Disks When Devices Are Nearly Full

If a device is nearly full, re-creating a clustered index does not balance data across devices. Instead, the device that is nearly full stores a small portion of the partition, and the other space allocations for the partition steals extents on other devices. Figure 33-15 shows a table with nearly full data devices.

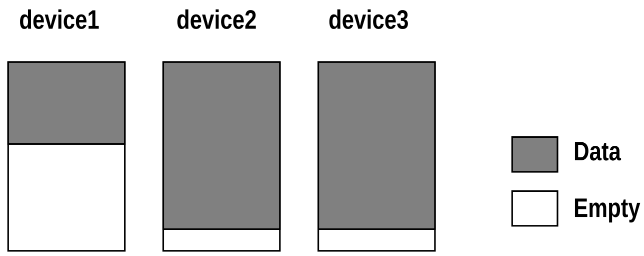


Figure 33-15: Partitions almost completely fill the devices

After adding devices and re-creating the clustered index, the result might be similar to the results shown in Figure 33-16.

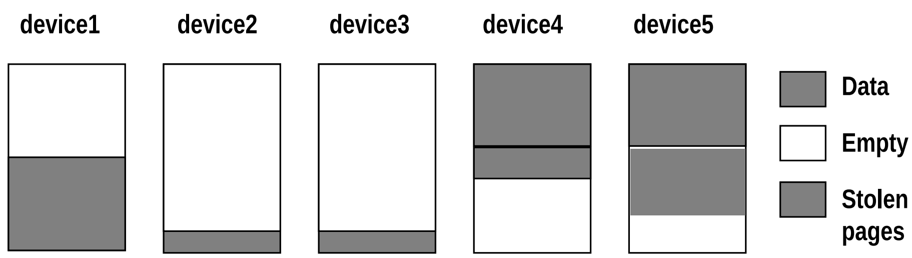


Figure 33-16: Extent stealing and unbalanced data distribution

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 does not solve the problem. Using bulk copy to copy the data out and back in again is the only sure solution to this form of imbalance.

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

Maintenance Issues and Partitioned Tables

Partitioned table maintenance activity requirements depend on the frequency and type of updates performed on the table.

Partitioned tables that require little maintenance 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 does 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% full. Adding space 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 33-31.

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.

34 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

Asynchronous prefetch improves performance by anticipating the pages required for certain well-defined classes of database activities whose access patterns are predictable. The I/O requests for these pages are issued before the query needs them so that most pages are in cache by the time query processing needs to access the page. Asynchronous prefetch can improve performance for:

- Sequential scans, such as table scans, clustered index scans, and covered nonclustered index scans
- Access via nonclustered indexes
- Some **dbcc** checks and **update statistics**
- Recovery

Asynchronous prefetch can improve the performance of queries that access large numbers of pages, such as decision support applications, as long as the I/O subsystems on the machine are not saturated.

Asynchronous prefetch cannot help (or may help only slightly) when the I/O subsystem is already saturated or when Adaptive Server is CPU-bound. It may be used in some OLTP applications, but to a much lesser degree, since OLTP queries generally perform fewer I/O operations.

When a query in Adaptive Server needs to perform a table scan, it:

- Examines the rows on a page and the values in the rows.
- Checks the cache for the next page to be read from a table. If that page is in cache, the task continues processing. If the page is not in cache, the task issues an I/O request and sleeps until the I/O completes.
- When the I/O completes, the task moves from the sleep queue to the run queue. When the task is scheduled on an engine, Adaptive Server examines rows on the newly fetched page.

This cycle of executing and stalling for disk reads continues until the table scan completes. In a similar way, queries that use a nonclustered index process a data page, issue the I/O for the next page referenced by the index, and sleep until the I/O completes, if the page is not in cache.

This pattern of executing and then waiting for I/O slows performance for queries that issue physical I/Os for large number of pages. In addition to the waiting time for the physical I/Os to complete, the task switches on and off the engine repeatedly. This task switching adds overhead to processing.

Improving Query Performance by Prefetching Pages

Asynchronous prefetch issues I/O requests for pages before the query needs them so that most pages are in cache by the time query processing needs to access the page. If required pages are already in cache, the query does not yield the engine to wait for the physical read. (It may still yield for other reasons, but it yields less frequently.)

Based on the type of query being executed, asynchronous prefetch builds a **look-ahead set** of pages that it predicts will be needed very soon. Adaptive Server defines different look-ahead sets for each processing type where asynchronous prefetch is used.

In some cases, look-ahead sets are extremely precise; in others, some assumptions and speculation may lead to pages being fetched that are never read. When only a small percentage of unneeded pages are read into cache, the performance gains of asynchronous prefetch far outweigh the penalty for the wasted reads. If the number of unused pages becomes large, Adaptive Server detects this condition and either reduces the size of the look-ahead set or temporarily disables prefetching.

Prefetching Control Mechanisms in a Multiuser Environment

When many simultaneous queries are prefetching large numbers of pages into a buffer pool, there is a risk that the buffers fetched for one query could be flushed from the pool before they are used.

Adaptive Server tracks the buffers brought into each pool by asynchronous prefetch and the number that are used. It maintains a per-pool count of prefetched but unused buffers. By default, Adaptive Server sets an asynchronous prefetch limit of 10 percent of each pool. In addition, the limit on the number of prefetched but unused buffers is configurable on a per-pool basis.

The pool limits and usage statistics act like a governor on asynchronous prefetch to keep the cache-hit ratio high and reduce unneeded I/O. Overall, the effect is to ensure that most queries experience a high cache-hit ratio and few stalls due to disk I/O sleeps.

The following sections describe how the look-ahead set is constructed for the activities and query types that use asynchronous prefetch. In some asynchronous prefetch optimizations, allocation pages are used to build the look-ahead set. For information on how allocation pages record information about object storage, see “Allocation Pages” on page 3-7.

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 the pages for 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 34-14 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 34-8.

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 the leaf 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 validate information on the allocation page. The look-ahead set for the **dbcc** operations that check allocation is similar to the look-ahead set for other sequential scans. When the scan enters a different allocation unit for the object, the look-ahead set is built from all the pages on the allocation unit that are used by the object.

checkdb and *checktable*

The **dbcc checkdb** and **dbcc checktable** commands check the page chains for a table, building the look-ahead set in the same way as other sequential scans.

If the table being checked has nonclustered indexes, they are scanned recursively, starting at the root page and following all pointers to the data pages. When checking the pointers from the leaf pages to the data pages, the **dbcc** commands use asynchronous prefetch in a way that is similar to nonclustered index scans. When a leaf-level index page is accessed, the look-ahead set is built from the page IDs of all the pages referenced on the leaf-level index page.

Look-Ahead Set Minimum and Maximum Sizes

The size of a look-ahead set for a query at a given point in time is determined by several factors:

- The type of query, such as a sequential scan or a nonclustered index scan

- The size of the pools used by the objects that are referenced by the query and the prefetch limit set on each pool
- The fragmentation of tables or indexes, in the case of operations that perform scans
- The recent success rate of asynchronous prefetch requests and overload conditions on I/O queues and server I/O limits

Table 34-1 summarizes the minimum and maximum sizes for different type of asynchronous prefetch usage.

Table 34-1: 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: • 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: • The number of unique page numbers on qualified rows on the leaf index page • The pool's prefetch limit
Recovery	Recovering a transaction	Maximum is the smaller of: • All of the data and index pages touched by a transaction undergoing recovery • The prefetch limit of the 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 17-99, and "max async i/os per server" on page 17-99 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 17-40 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 39-77.

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 on allpages-locked tables
- Heavy cache utilization by multiple users

Page Chain Fragmentation

Adaptive Server’s page allocation mechanism strives to keep pages that belong to the same object close to each other in physical storage by allocating new pages on an extent already allocated to the object and by allocating new extents on allocation units already used by the object.

However, as pages are allocated and deallocated, page chains on data-only-locked tables can develop kinks. Figure 34-1 shows an example of a kinked page chain between extents in two allocation units.

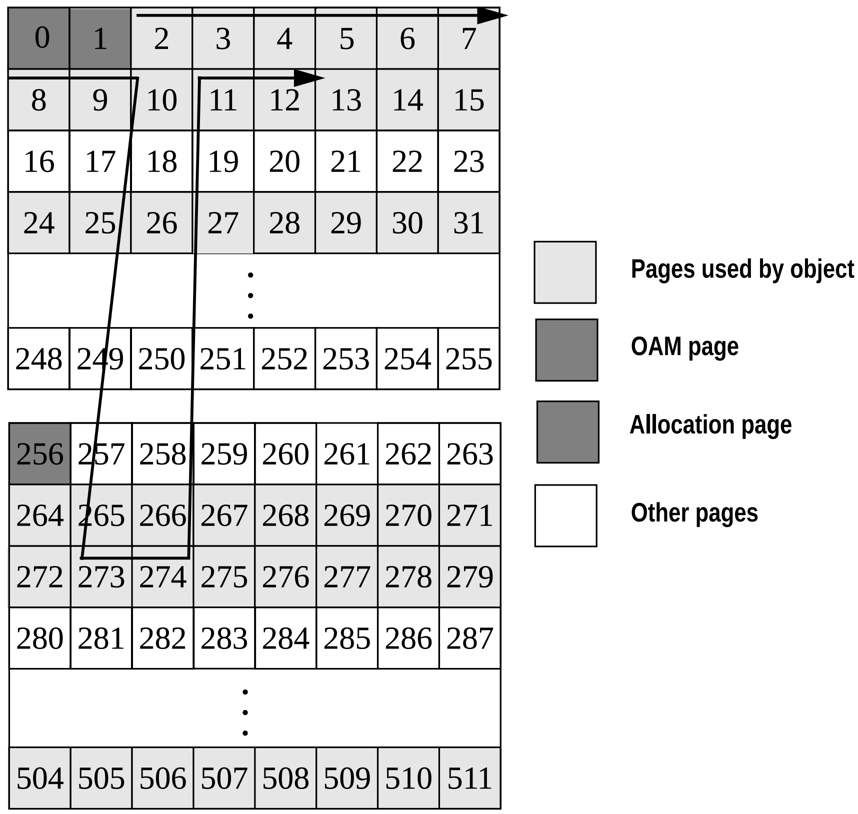


Figure 34-1: A kink in a page chain crossing allocation units

In Figure 34-1, when a scan first needs to access a page from allocation unit 0, it checks the allocation page and issues asynchronous I/Os for all the pages used by the object it is scanning, up to the limit set on the pool. As the pages become available in cache, the query processes them in order by following the page chain. When the scan reaches page 10, the next page in the page chain, page 273, belongs to allocation unit 256.

When page 273 is needed, allocation page 256 is checked, and asynchronous prefetch requests are issued for all the pages in that allocation unit that belong to the object.

When the page chain points back to a page in allocation unit 0, there are two possibilities:

- The prefetched pages from allocation unit 0 are still in cache, and the query continues processing with no unneeded physical I/Os.
- The 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 two. It uses the ratio between the count of hops and the prefetched pages to reduce the size of the look-ahead set, so fewer I/Os are issued.
- The count of prefetched but unused pages in the pool is likely to be high, so asynchronous prefetch may be temporarily discontinued or reduced, based on the pool's limit.

Tuning Goals for Asynchronous Prefetch

Choosing optimal pool sizes and prefetch percentages for buffer pools can be key to achieving improved performance with asynchronous prefetch. When multiple applications are running concurrently, a well-tuned prefetching system balances pool sizes and prefetch limits to accomplish these goals:

- Improved system throughput
- Better performance by applications that use asynchronous prefetch
- No performance degradation in applications that do not use asynchronous prefetch

Configuration changes to pool sizes and the prefetch limits for pools are dynamic, allowing you to make changes to meet the needs of varying workloads. For example, you can configure asynchronous prefetch for good performance during recovery or **dbcc** checking and reconfigure afterward without needing to restart Adaptive Server. See "Setting Asynchronous Prefetch Limits for Recovery" on page 34-14 and "Setting Asynchronous Prefetch Limits for dbcc" on page 34-14 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, the default value for **global async prefetch limit** is 10 (percent.) For more information, see "global async prefetch limit" on page 17-28 of the *System Administration Guide*.

- A per-pool override, set with **sp_poolconfig**. To see the limits set for each pool, use **sp_cacheconfig**. For more information, see "Changing the Asynchronous Prefetch Limit for a Pool" on page 15-23 of the *System Administration Guide*.

Changing asynchronous prefetch limits takes effect immediately, and does not require a reboot. Both the global and per-pool limits can also be configured in the configuration file.

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 poll 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 buffer pools can become higher with parallel queries. With serial queries operating on the same pools, it is safe to assume that queries are issued at slightly different times and that the queries are in different stages of execution: Some are accessing pages are already in cache, and others are waiting on I/O.

Parallel execution places different demands on buffer pools, depending on the type of scan and the degree of parallelism. Some parallel queries are likely to issue a large number of prefetch requests simultaneously.

Hash-Based Table Scans

Hash-based table scans on allpages-locked tables have multiple worker processes 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 table scans do not place higher demands on the pools than they place on serial processes. Since the multiple processes may read the pages much more quickly than a serial process does, they change the status of the pages from unused to used more quickly.

Partition-Based Scans

Partition-based scans are more likely to create additional demands on pools, since multiple worker processes may be performing asynchronous prefetching on different allocation units. On partitioned tables on multiple devices, the per-server and per-engine I/O limits are less likely to be reached, but the per-pool limits are more likely to limit prefetching.

Once a parallel query is parsed and compiled, it launches its worker processes. If a table with 4 partitions is being scanned by 4 worker processes, each worker process attempts to prefetch all the pages in its first allocation unit. For the performance of this single query, the most desirable outcome is that the size and limits on the 16K pool are sufficiently large to allow 124 (31*4) asynchronous prefetch requests, so all of the requests succeed. Each of the worker processes scans the pages in cache quickly, moving onto new allocation units and issuing more prefetch requests for large numbers of pages.

Special Settings for Asynchronous Prefetch Limits

You may want to change asynchronous prefetch configuration temporarily for specific purposes, including:

- Recovery
- **dbcc** operations that use asynchronous prefetch

Setting 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 original 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 for all pages-locked tables and the leaf levels of indexes develop kinks as data modifications take place on the table. In general, newly created tables have few kinks. Tables where updates, deletes and inserts that have caused page splits, new page allocations, and page deallocations are likely to have cross-allocation unit page chain kinks. If more than 10 to 20 percent of the original rows in a table have been modified, you should 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 allpages-locked heaps, page allocation is generally sequential, unless pages are deallocated by deletes that remove all rows from a page. These pages may be reused when additional space is allocated to the object. You can create a clustered index (and drop it, if you want the table stored as a heap) or bulk copy the data out, truncate the table, and copy the data in again. Both activities compress the space used by the table and eliminate page-chain kinks.

Eliminating Kinks in Clustered Index Tables

For clustered indexes, page splits and page deallocations can cause page chain kinks. Rebuilding clustered indexes does not necessarily eliminate all cross-allocation page linkages. Use **fillfactor** for clustered indexes where you expect growth, to reduce the number of kinks resulting from data modifications.

Eliminating Kinks in Nonclustered Indexes

If your query mix uses covered index scans, dropping and re-creating nonclustered indexes can improve asynchronous prefetch performance, once the leaf-level page chain becomes fragmented.

Performance Monitoring and Asynchronous Prefetch

The output of **statistics io** reports the number physical reads performed by asynchronous prefetch and the number of reads performed by normal asynchronous I/O. In addition, **statistics io** reports the number of times that a search for a page in cache was found by the asynchronous prefetch without holding the cache spinlock. See Chapter 16, “Reporting Physical and Logical I/O Statistics,” for more information.

sp_sysmon 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 for 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 39-71 and “Disk I/O Management” on page 39-92 for more information.

35 *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 for storing temporary tables and indexes created by users

Many applications use stored procedures that create tables in *tempdb* to expedite complex joins or to perform other complex data analysis that is not easily performed in a single step.

This chapter contains the following sections:

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(col1)
```

Regular User Tables

You can create regular user tables in *tempdb* by specifying the database name in the command that creates the table:

```
create table tempdb..temptable
```

or:

```
select select_list
    into tempdb..temptable
```

Regular user tables in *tempdb:*

- Can persist across sessions
- Can be used by bulk copy operations
- Can be shared by granting permissions on them
- Must be explicitly dropped by the owner (otherwise, they are removed when Adaptive Server is restarted)

You can create indexes in *tempdb* on permanent temporary tables:

```
create index tempix on tempdb..temptable(col1)
```

Worktables

Worktables are automatically created in *tempdb* by Adaptive Server for merge joins, sorts, and other internal server processes. These tables:

- Are never shared
- Disappear as soon as the command completes

Initial Allocation of *tempdb*

When you install Adaptive Server, *tempdb* is 2MB, and is located completely on the master device, as shown in Figure 35-1. 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.

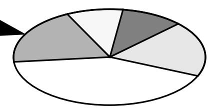


Figure 35-1: tempdb default allocation

Use **sp_helpdb** to see the size and status of *tempdb*. The following example shows *tempdb* defaults at installation time:

```
sp_helpdb tempdb
name      db_size  owner  dbid   created     status
--------- -------- ------ ------ ----------- --------------------
tempdb    2.0 MB   sa     2     May 22, 1999 select into/bulkcopy

device_frag  size    usage        free kbytes
----------- -------- ----------- ---------
master       2.0 MB  data and log 1248
```

Sizing *tempdb*

tempdb needs to be big enough to handle the following processes for every concurrent Adaptive Server user:

- Worktables for merge joins
- Worktables that are created for **distinct, group by**, and **order by**, for reformatting, and for the OR strategy, and for materializing some views and subqueries
- Temporary tables (those created with “#” as the first character of their names)
- Indexes on temporary tables
- Regular user tables in *tempdb*
- Procedures built by dynamic SQL

Some applications may perform better if you use temporary tables to split up multitable joins. This strategy is often used for:

- Cases where the optimizer does not choose a good query plan for a query that joins more than 4 tables
- Queries that join a very large number of tables
- Very complex queries
- Applications that need to filter data as an intermediate step

You might also use *tempdb* to:

- Denormalize several tables into a few temporary tables
- Normalize a denormalized table in order to do aggregate processing

For most applications, make *tempdb* 20 to 25% of the size of your user databases to provide enough space for these uses.

Placing *tempdb*

Keep *tempdb* on separate physical disks from your critical application databases 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. After you expand *tempdb* onto additional devices, drop the master device from the *system*, *default*, and *logsegment s*egments.

Although you can expand *tempdb* on the same device as the *master* database, it is usually preferable to use separate devices. Also, remember that logical devices are mirrored using Adaptive Server mirroring, 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.

Dropping the master Device from *tempdb* Segments

By default, the *system*, *default*, and *logsegment* segments for *tempdb* include its 2MB allocation on the master device. When you allocate new devices to *tempdb*, they automatically become part of all three segments. Once you allocate a second device to *tempdb*, you can drop the master device from the *default* and *logsegment* segments. This way, you can be sure that the worktables and other temporary tables in *tempdb* do not contend with other uses on the master device.

To drop the master device from the segments:

1. Alter *tempdb* onto another device, if you have not already done so. For example:

   alter database tempdb on tune3 = 20

2. Issue a **use tempdb** command, and then drop the master device from the segments:

   sp_dropsegment "default", tempdb, master

   sp_dropdegment system, tempdb, master

   sp_dropdegment 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 35-2, you can take advantage of parallel query performance for some temporary tables or worktables.

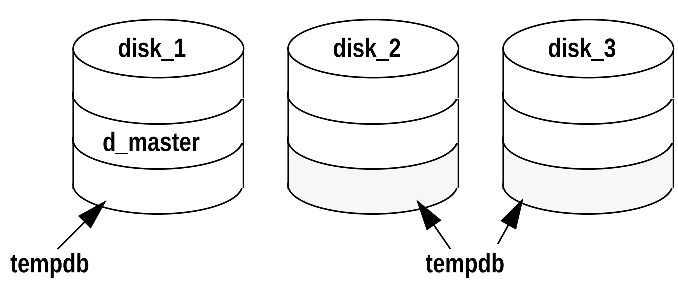


Figure 35-2: tempdb spanning disks

Binding *tempdb* to Its Own Cache

Under normal Adaptive Server use, *tempdb* makes heavy use of the data cache as temporary tables are created, populated, and then dropped.

Assigning *tempdb* to its own data cache:

- Keeps the activity on temporary objects from flushing other objects out of the default data cache
- Helps spread I/O between multiple caches

See "Examining Cache Needs for tempdb" on page 32-23 for more information.

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 15-6 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

Creating or dropping temporary tables and their indexes can cause lock contention on the system tables in *tempdb*. When users create tables in *tempdb*, information about the tables must be stored in system tables such as *sysobjects*, *syscolumns*, and *sysindexes*. If multiple user processes are creating and dropping tables in *tempdb*, heavy contention can occur on the system tables. Worktables created internally do not store information in system tables.

If contention for *tempdb* system tables is a problem with applications that must repeatedly create and drop the same set of temporary tables, try creating the tables at the start of the application. Then use **insert...select** to populate them, and **truncate table** to remove all the data rows. Although **insert...select** requires logging and is slower than **select into**, it can provide a solution to the locking problem.

Minimizing Logging in *tempdb*

Even though the **trunc log on checkpoint** database option is turned on in *tempdb*, changes to *tempdb* are still written to the transaction log. You can reduce log activity in *tempdb* by:

- Using **select into** instead of **create table** and **insert**
- Selecting only the columns you need into the temporary tables

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 35-3. This applies to both temporary tables and regular user tables.

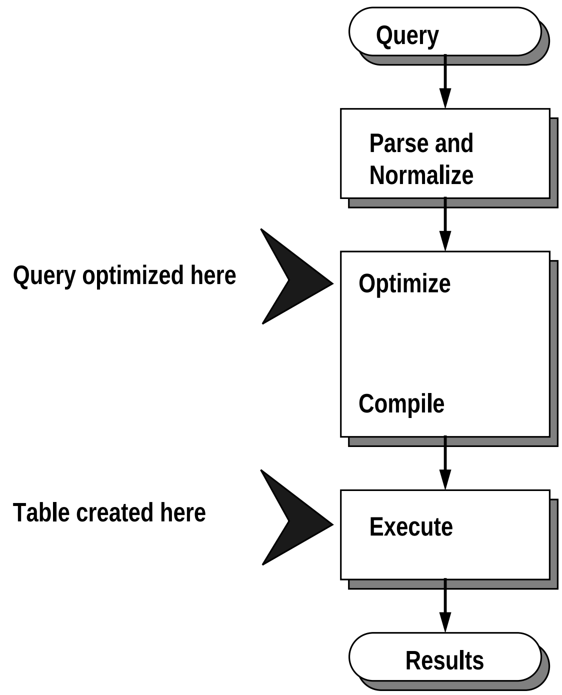


Figure 35-3: Optimizing and creating temporary tables

The optimizer assumes that any such table has 10 data pages and 100 rows. If the table is really large, this assumption can lead the optimizer to choose a suboptimal query plan.

These two techniques can improve the optimization of temporary tables:

- Creating indexes on temporary tables
- Breaking complex use of temporary tables into multiple batches or procedures to provide information for the optimizer

Creating Indexes on Temporary Tables

You can define indexes on temporary tables. In many cases, these indexes can improve the performance of queries that use *tempdb*. The optimizer uses these indexes just like indexes on ordinary user tables. The only requirements are:

- The table must contain data when the index is created. If you create the temporary table and create the index on an empty table, Adaptive Server does not create column statistics such as histograms and densities. If you insert data rows after creating the index, the optimizer has incomplete statistics.
- The index must exist 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 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 *base_proc* procedure calls the *select_proc* procedure, the optimizer can determine the size of the table:

```
create proc select_proc
as
    select *
        from tab, #huge_result where ...

create proc base_proc
as
    select *
        into #huge_result
        from ...
    exec select_proc
```

If the processing for *#huge_result* requires multiple accesses, joins, or other processes, such as looping with **while**, creating an index on

#huge_result may improve performance. Create the index in *base_proc* so that it is available when *select_proc* is optimized.

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*.

36 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?

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 39, "Monitoring Performance with sp_sysmon." Pay special attention to the output in "Network I/O Management" on page 39-96.

How Adaptive Server Uses the Network

All client/server communication occurs over a network via packets. Packets contain a header and routing information, as well as the data they carry. Adaptive Server was one of the first database systems to be built on a network-based client/server architecture. Clients initiate a connection to the server. The connection sends client requests and server responses. Applications can have as many connections open concurrently as they need to perform the required task. The protocol used between the client and server is known as the Tabular Data Stream™ (TDS), which forms the basis of communication for many Sybase products.

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 17, "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 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
- **select** statements with large result sets

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 36-1 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.

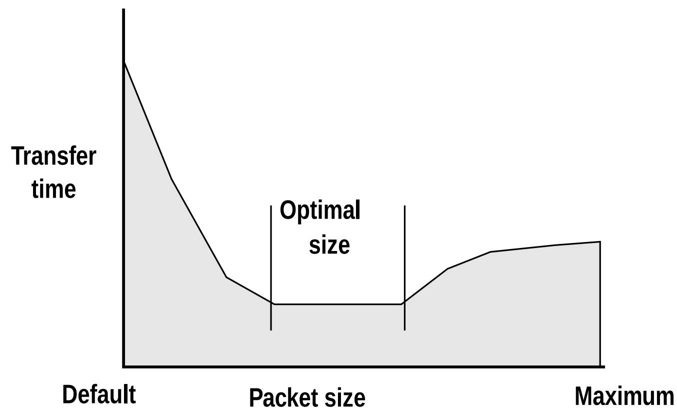


Figure 36-1: Packet sizes and performance

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:

isql -A*size*

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 commands 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.

You may be able to reduce network overhead by turning off "doneinproc" packets. For more information, see "Reducing Packet Overhead" on page 39-100.

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 36-2. In many cases, this can also reduce the disk I/O load.

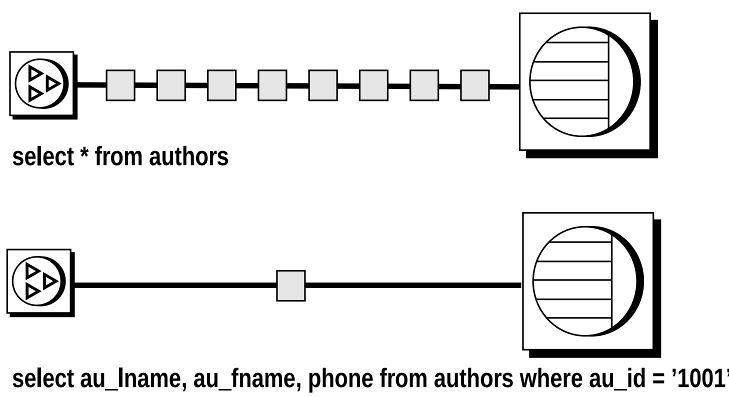


Figure 36-2: Reducing network traffic by filtering data at the server

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 36-1 shows the characteristics of some network types.

Table 36-1: 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 36-3, before digging more deeply into the database system to solve an abnormal response time problem.

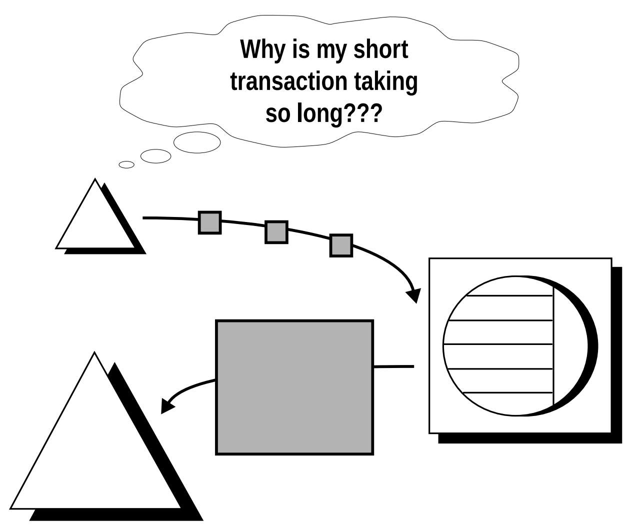


Figure 36-3: Effects of long transactions on other users

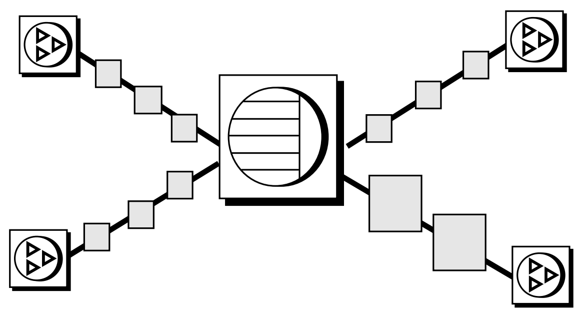
Guidelines for Improving Network Performance

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 36-4.
- 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

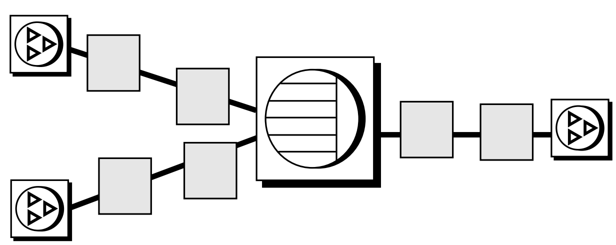


Figure 36-4: Match network packet sizes to application mix

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 36-5.

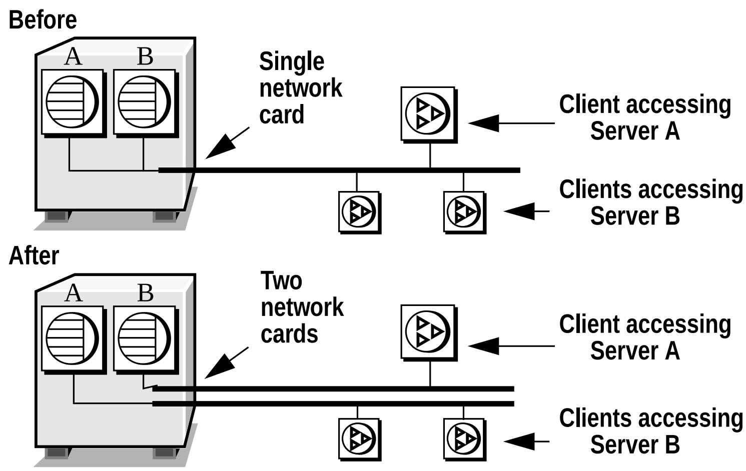


Figure 36-5: Isolating heavy network users

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.

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 36-6. 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.

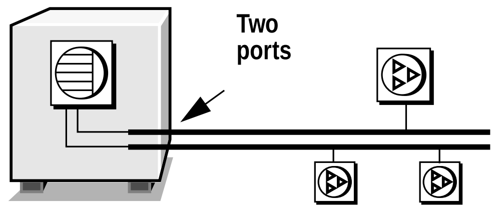


Figure 36-6: Configuring multiple network ports

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

37 How Adaptive Server Uses Engines and CPUs

Adaptive Server's multithreaded architecture is designed for high performance in both uniprocessor and multiprocessor systems. This chapter describes how Adaptive Server uses engines and CPUs to fulfill client requests and manage internal operations. It introduces Adaptive Server's use of CPU resources, describes the Adaptive Server 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

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.

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. 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 11-3.
6. Adaptive Server executes the task. If the query was executed in parallel, the task merges the results of the subtasks.
7. The task returns the results to the client, using TDS packets.

For each new user connection, Adaptive Server allocates a private data storage area, a dedicated stack, and other internal data structures. It uses the stack to keep track of each client task’s state during processing, and it uses synchronization mechanisms such as 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 processes use. 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 37-1 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.

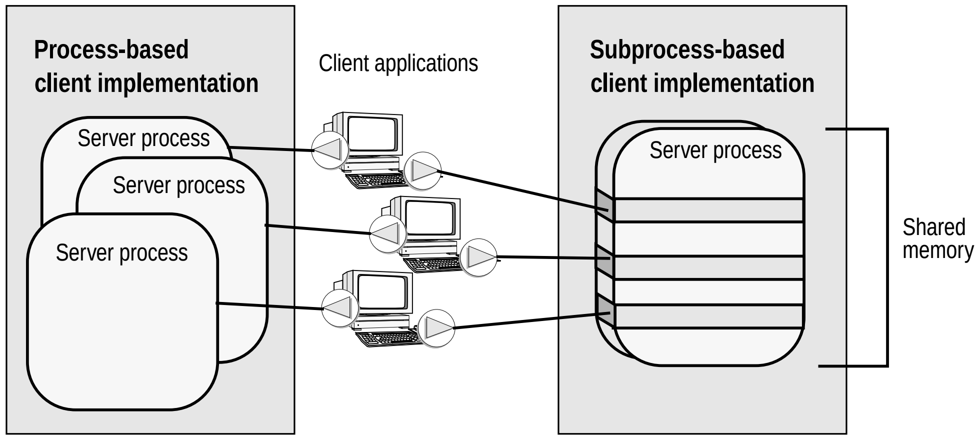


Figure 37-1: Process vs. subprocess architecture

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 37-10 expands on this discussion to show how an Adaptive Server system with multiple CPUs processes client requests.

Scheduling Engines to the CPU

Figure 37-2 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.

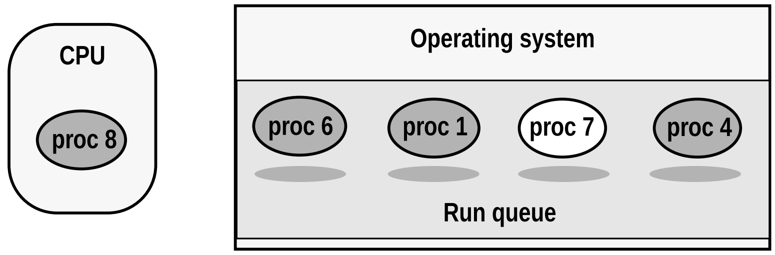


Figure 37-2: Processes queued in the run queue for a single CPU

In a multitasking environment, multiple processes or subprocesses execute concurrently, alternately sharing CPU resources. Figure 37-3 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.

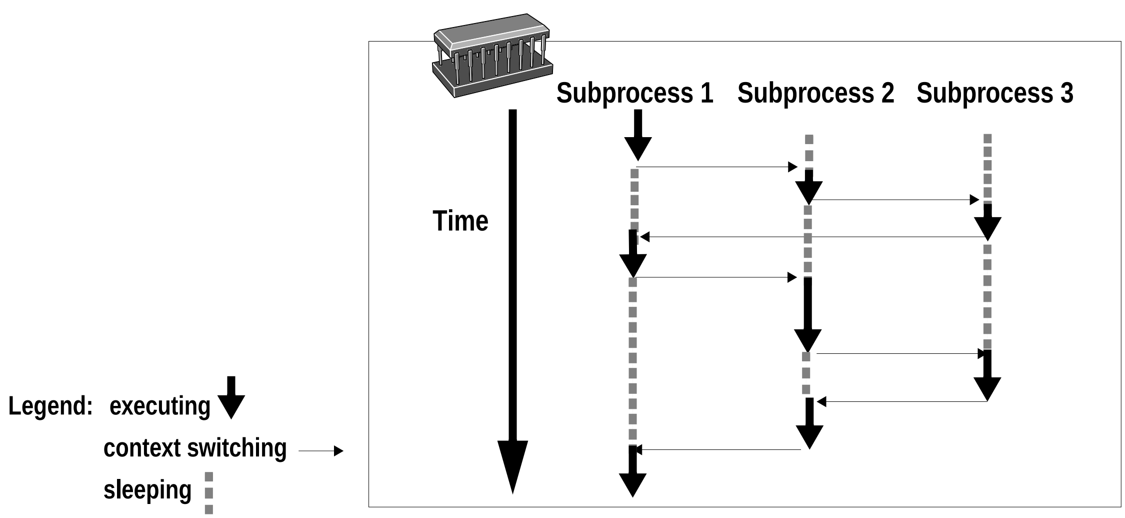


Figure 37-3: Multithreaded processing

Scheduling Tasks to the Engine

Figure 37-4 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 37-2 on page 37-5) 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 a page that is locked by another task, or it needs to perform a slow job such as disk I/O or network I/O. The task is put to sleep, waiting for the resource.
- The task runs for a configurable period of time and reaches a yield point. Then the task relinquishes the engine, and the next process in the queue starts to run. "Scheduling Client Task Processing Time" on page 37-7 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 37-4 also shows the sleep queue with two sleeping tasks, as well as other objects in shared memory. Tasks are put to sleep while they are waiting for resources or for the results of a disk I/O operation.

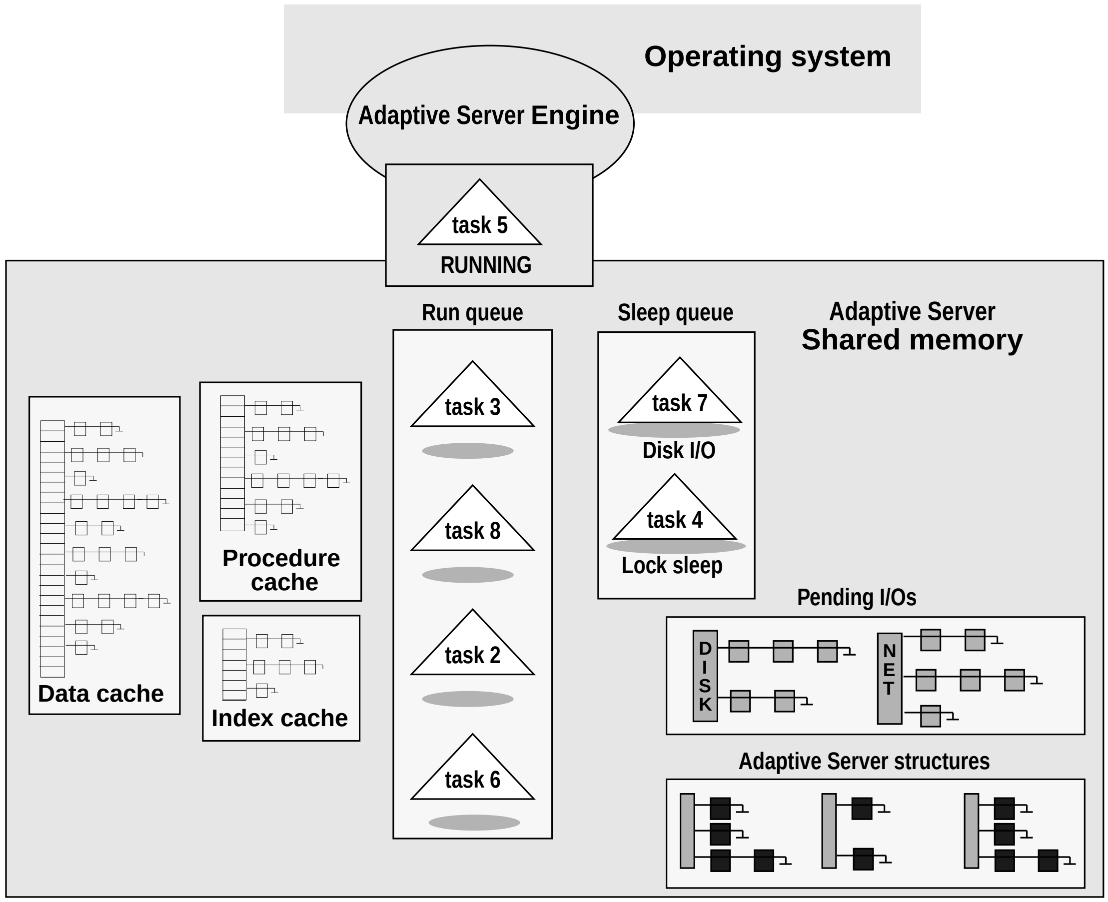


Figure 37-4: Tasks queue up for the Adaptive Server engine

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** prevents executing tasks from monopolizing engines during execution. The scheduler allows a task to execute on an Adaptive Server engine for a maximum amount of time that is equal to the **time slice** and **cpu grace time** values combined, as illustrated in Figure 37-5, using default times for **time slice** 100 milliseconds, 1/10 of a second, or equivalent to one clock tick) and **cpu grace time** (500 clock ticks, or 50 seconds).

Figure 37-5 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**, when the task does not hold a vital resource such as a spinlock.

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

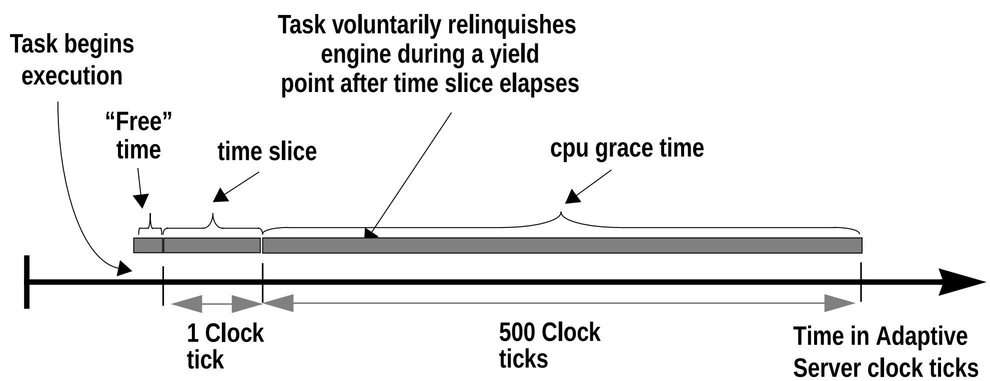


Figure 37-5: Task execution time schedule when other tasks are waiting

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 a yield point during grace time, Adaptive Server grants the task another processing interval as shown Figure 37-6.

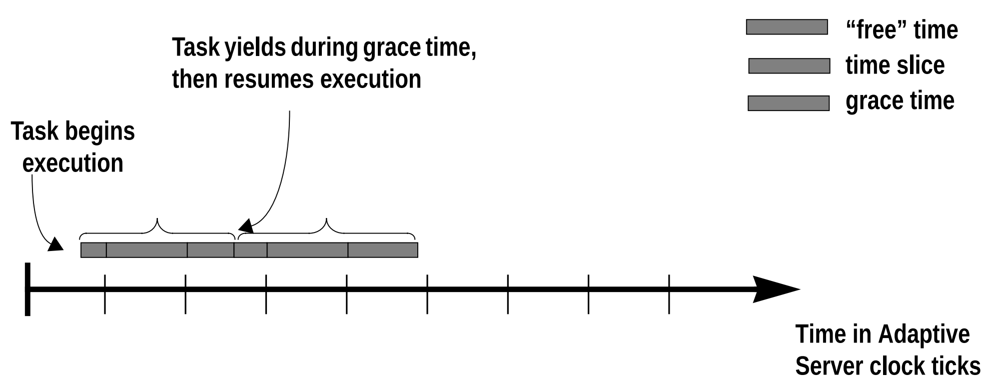


Figure 37-6: Task execution time schedule when no other tasks are waiting

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 37-7 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.

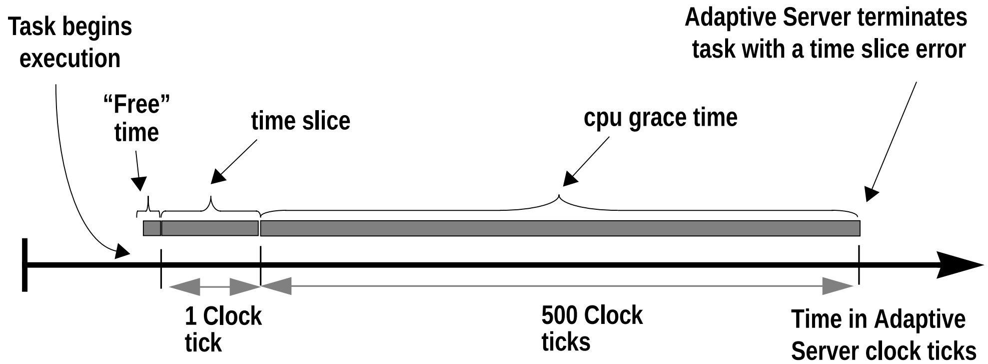


Figure 37-7: A task fails to relinquish the engine within the scheduled time

Maintaining CPU Availability During Idle Time

When Adaptive Server has no tasks to run, it loops (holds the CPU), looking for executable tasks. The configuration parameter **runnable process search count** controls the number of times that Adaptive Server loops. With the default value of 2000, Adaptive Server loops 2000 times, looking for incoming client requests, completed disk I/Os, and new tasks in the run queue. If there is no activity for the duration of **runnable process search count**, Adaptive Server relinquishes the CPU to the operating system.

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

Use **sp_sysmon** to determine how **runnable process search count** affects Adaptive Server's use of CPU cycles, engine yields to the operating system, and blocking network checks. See "Engine Busy Utilization" on page 39-12 and "Network Checks" on page 39-14.

Tuning Scheduling Parameters

The default value for the **time slice** parameter is 100 clock milliseconds, and there is seldom any reason to change it. The default value for **cpu grace time** is 500 clock ticks. If **time slice** is set too low, an engine 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 want to increase the amount of time that CPU-intensive applications run on an engine before yielding, you can assign execution attributes to specific logins, applications, or stored procedures. See Chapter 38, “Distributing Engine Resources Between Tasks,” for more information.

sp_sysmon to determine how many times tasks yield voluntarily. See “Voluntary Yields” on page 39-23.

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 128 processors
- Very high throughput

Scheduling Engines to CPUs

In a system with multiple CPUs, multiple processes can run concurrently. Figure 37-8 represents Adaptive Server engines as the nonshaded ovals waiting in the operating system run queue for processing time on one of three CPUs. It shows two Adaptive Server engines, proc 3 and proc 8, being processed simultaneously.

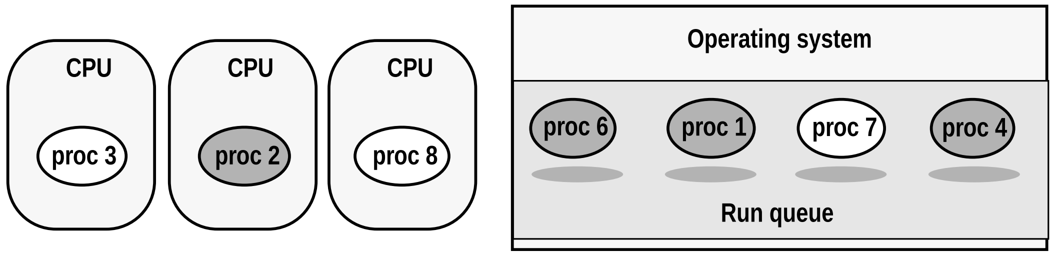


Figure 37-8: Processes queued in the OS run queue for multiple CPUs

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 37-5. The difference is that in the SMP environment:

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

Multiple Network Engines

Each Adaptive Server engine handles the network I/O for its connections. Engines are numbered sequentially, starting with engine 0. When a user logs in to Adaptive Server, the task is assigned in round-robin fashion to one of the engines that will serve as its

network engine. This engine handles the login to establish packet size, language, character set, and other login settings. All network I/O for a task is managed by its network engine until the task logs out.

Task Priorities and Run Queues

At certain times, Adaptive Server increases the priority of some tasks, especially if they are holding an important resource or have had to wait for a resource. In addition, logical process management allows you to assign priorities to logins, procedures, or applications using **sp_bindexeclass** and related system procedures. See Chapter 38, “Distributing Engine Resources Between Tasks,” for more information on performance tuning and task priorities.

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

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

A Processing Scenario

Figure 37-9 shows a conceptual representation of the SMP subsystems, which consist of clients, disks, the operating system, multiple CPUs, and the Adaptive Server executable. It shows these 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
- The global and engine run queues for processes that are ready to execute

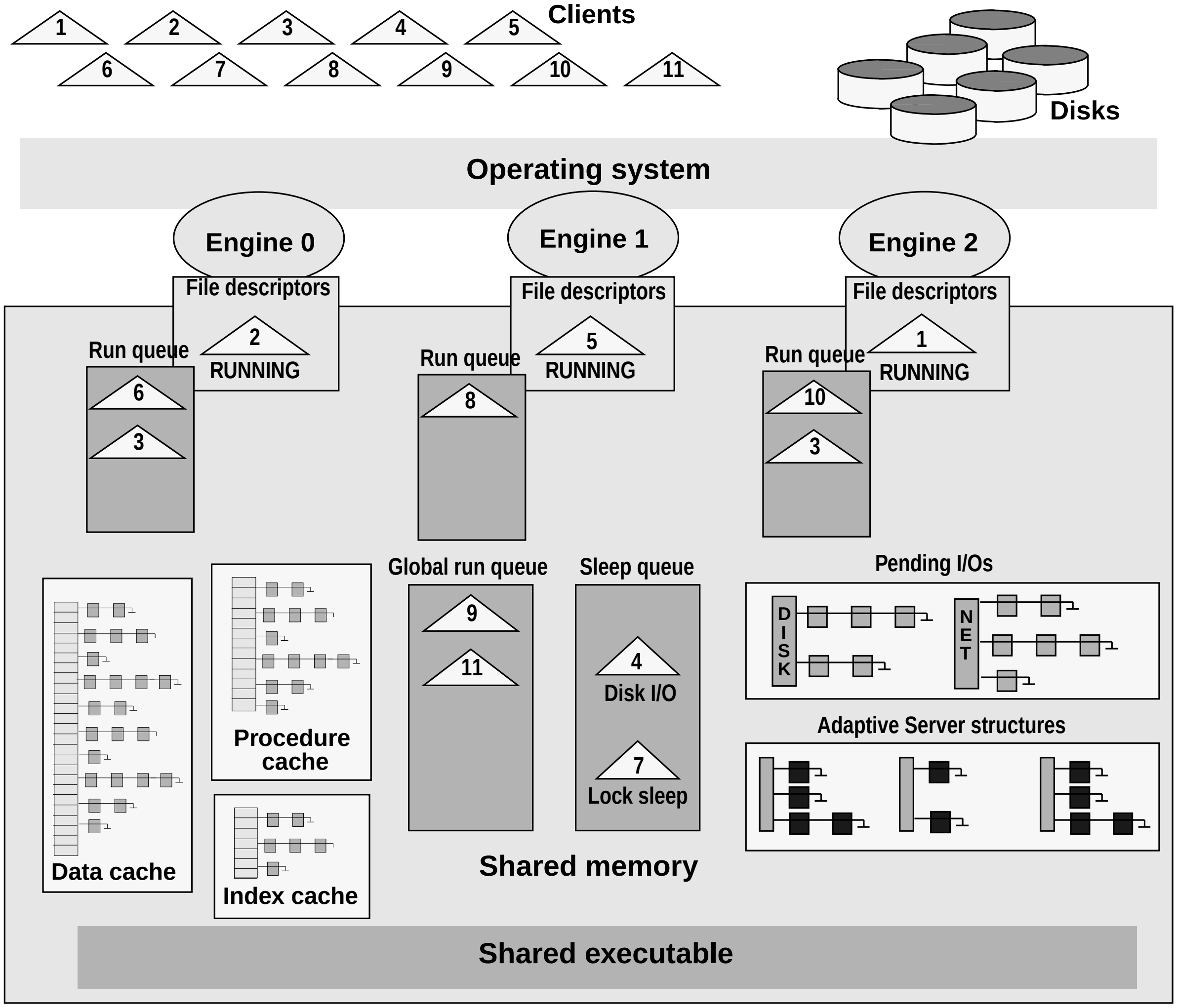


Figure 37-9: Adaptive Server task management in the SMP environment

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

Assigning a Network Engine During Login

1. When a connection logs in to Adaptive Server, the connection is assigned to an engine that will manage its network I/O. This engine then handles the login. The engine assigns a task structure and establishes packet size, language, character set, and other login settings. In Figure 37-9, Engine 2 is the network

engine for task 5. Task 5 sleeps while it waits for the client to send a request.

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 its run queue.

Fulfilling a Client Request

4. When task 5 becomes first in the queue, Engine 2 parses, compiles, and begins executing the steps defined in the task's query plan.

Performing Disk I/O

5. If the task needs to access a page locked by another user, it is put to sleep until the page is available. After such a wait, the task's priority is increased, and it is placed in the global run queue so that any engine can run it. Figure 37-9 shows the task executing on Engine 1.

Performing Network I/O

6. When the task needs to return results to the user, the engine on which it is executing issues the network I/O request, and puts the tasks to sleep on a network write.
7. Engine 2 checks once each clock tick to determine whether the network I/O has completed. When the I/O has completed, the task is placed on the run queue for the engine to which it is affinitied, or the global run queue.

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 and performing other maintenance tasks. 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. However, setting **housekeeper free write percent** too high can degrade performance. If you want to increase the value, increase by only 1 or 2 percent each time.

A **dbcc tune** option, **deviochar**, controls the size of batches that the housekeeper can write to disk at one time. See "Increasing the Housekeeper Batch Limit" on page 39-91.

Changing the Percentage by Which Writes Can Be Increased

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

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

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
```

◆ ***WARNING!***

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

Allowing the Housekeeper Task to Work Continuously

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

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

The "Recovery Management" section of **sp_sysmon** shows checkpoint information to help you determine the effectiveness of the housekeeper. See "Recovery Management" on page 39-88.

Measuring CPU Usage

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

Single CPU Machines

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

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

Using *sp_monitor* to Measure CPU Usage

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

```
last_run                    current_run                 seconds
-------------------------- -------------------------- -----------
       Jul 28 1999 5:25PM        Jul 28 1999 5:31PM         360

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

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

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

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

Using *sp_sysmon* to Measure CPU Usage

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

The "CPU Yields by Engine" section displays information about how often the engine yielded to the operating system during the interval. See Chapter 39, "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 documented in the Adaptive Server installation and configuration guides.

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

Determining When to Configure Additional Engines

When you are determining whether to add additional engines, the major factors to 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 for the following sections in Chapter 39, "Monitoring Performance with sp_sysmon." In particular, study the lines or sections in the output that may reveal points of contention:

- "Logical Lock Contention" on page 39-25.
- "Address Lock Contention" on page 39-25.

- “ULC Semaphore Requests” on page 39-49.
- “Log Semaphore Requests” on page 39-50.
- “Page Splits” on page 39-55.
- “Lock Summary” on page 39-65.
- “Cache Spinlock Contention” on page 39-80.
- “I/Os Delayed By” on page 39-93.

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

Taking Engines Offline

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

dbcc engine(offline , [*enginenum*])

dbcc engine("online")

If *enginenum* is not specified, the highest-numbered engine is taken offline. For more information, see “Taking Engines Offline with dbcc engine” on page 16-5 of the *System Administration Guide*.

Enabling Engine-to-CPU Affinity

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

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

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

The syntax is:

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

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

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

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

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

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

results in the following affinity:

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

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

Engine	CPU
0	2
1	3
2	0

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

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

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

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

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

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

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

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

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

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

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

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

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

Multiprocessor Application Design Guidelines

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

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

Multiple Indexes

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

For information about the effects of index maintenance on performance, see “Index Management” on page 39-51.

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 39-92 for information about **sp_sysmon** reports on disk utilization.

Adjusting the *fillfactor* for *create index* Commands

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

Transaction Length

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

Temporary Tables

Temporary tables (tables in *tempdb*) do not cause contention, because they are associated with individual users and are not shared. However, if multiple user processes use *tempdb* for temporary objects, there can be some contention on the system tables in *tempdb*. See “Temporary Tables and Locking” on page 35-7 for information on ways to reduce contention.

38 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 37, "How Adaptive Server Uses Engines and CPUs."

This chapter discusses the following topics:

Using Execution Attributes to Manage Preferred Access to Resources

Most performance-tuning techniques give you control at the broad system or the specific query level. Adaptive Server also gives you control over the relative performance of simultaneously running tasks. Unless you have unlimited resources, the need for control at the task level is greater in parallel execution environments because there is more competition for limited resources.

You can use system procedures to assign **execution attributes** that indicate which tasks should be given preferred access to resources. The Logical Process Manager uses the execution attributes when it assigns priorities to tasks and tasks to engines. Execution attributes also affect how long a process can use an engine each time the process runs. In effect, assigning execution attributes lets you suggest to Adaptive Server how to distribute engine resources between client applications, logins, and stored procedures in a mixed workload environment.

Each client application or login can initiate many Adaptive Server tasks. In a single-application environment, you can distribute resources at the login and task levels 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 *EC1*, Adaptive Server considers it to be critical and tries to give it preferred access to engine resources. Any execution object associated with *EC3* is considered to be least critical and does not receive engine resources until execution objects associated with *EC1* and *EC2* are executed. By default, execution objects have *EC2* attributes.

To change an execution object's execution class from the *EC2* default, use **sp_bindexeclass** (described in "Assigning Execution Classes" on page 38-7).

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_addexeclass** 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_addexeclass DS, LOW, 0, ANYENGINE
```

You associate a user-defined execution class with an execution object using **sp_bindexeclass** 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 38-1. You specify the mix of attribute values for user-defined execution classes when you create them, using the system procedure **sp_addexeclass**.

Table 38-1: Fixed-attribute composition of predefined execution classes

Execution Class Level	Base Priority Attribute*	Time Slice Attribute **	Engine Affinity Attribute***
EC1	High	Time slice > t	None
EC2	Medium	Time slice = t	None
EC3	Low	Time slice < t	Engine with the highest engine ID number
	*See “Base Priority” on page 38-4	** See “Time Slice” on page 38-5	*** See “Task-to-Engine Affinity” on page 38-6

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

Base Priority

Base priority is the priority assigned to a task when it is created. The values are “high,” “medium,” and “low.” There is a run queue for each priority for each engine, and the global run queue also has a queue for each priority. Figure 38-1 shows tasks in the run queue that correspond to the three levels of base priority.

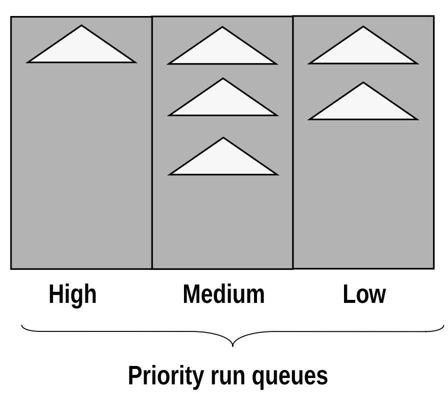


Figure 38-1: Tasks queued in the three priority run queues

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

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

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

Time Slice

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

As shown in Table 38-1 on page 38-4, the time slice attribute is different for each predefined execution class. *EC1* has the longest time slice value, *EC3* has the shortest time slice value, and *EC2* has a time slice value that is between the values for *EC1* and *EC3*.

More precisely, the time period that each task is allowed to run is based on the value for the **time slice** configuration parameter, as described in "Scheduling Client Task Processing Time" on page 37-7. Using default values for configuration parameters, *EC1* execution objects may run for double the time slice value; the time slice of an *EC2* execution object is equivalent to the configured value; and an *EC3* execution object yields at the first yield point it encounters, often not running for an entire time slice.

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

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

Task-to-Engine Affinity

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

- Associate less critical execution objects with a defined group of engines to restrict the object to a subset of the total number of engines. This reduces processor availability for those objects. The more critical execution objects can execute on any Adaptive Server engine, so performance for them improves because they have the benefit of the resources that the less critical ones are deprived of.
- Associate more critical execution objects with a defined group of engines to which less critical objects do not have access. This ensures that the critical execution objects have access to a known amount of processing power.

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

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

➤ ***Note***

The engine affinity attribute is not used for stored procedures.

Setting Execution Class Attributes

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

Table 38-2: System procedures for managing execution object precedence

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

See the *Adaptive Server Reference Manual* for complete descriptions of the system procedures in Table 38-2.

Assigning Execution Classes

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

The syntax for the **sp_bindexeclass** is:

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

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

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

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

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

Engine Groups and Establishing Task-to-Engine Affinity

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

The example uses **sp_addengine** and **sp_addexeclass**.

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

```
sp_addengine engine_number, engine_group
```

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

```
sp_addexeclass class_name, base_priority,
               time_slice, engine_group
```

The steps are:

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

   ```
   sp_addengine 3, DS_GROUP
   ```

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

   ```
   sp_addengine 4, DS_GROUP
   ```

   ```
   sp_addengine 5, DS_GROUP
   ```

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

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

```
sp_addexeclass 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
```

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

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

Figure 38-2 illustrates the associations in this scenario:

- Logins "mgr1", "mgr2", and "mgr3" have affinity to the *DS* engine group consisting of engines 3, 4, and 5.
- Logins "ts1", "ts2", "ts3", and "ts4" can use all six Adaptive Server engines.

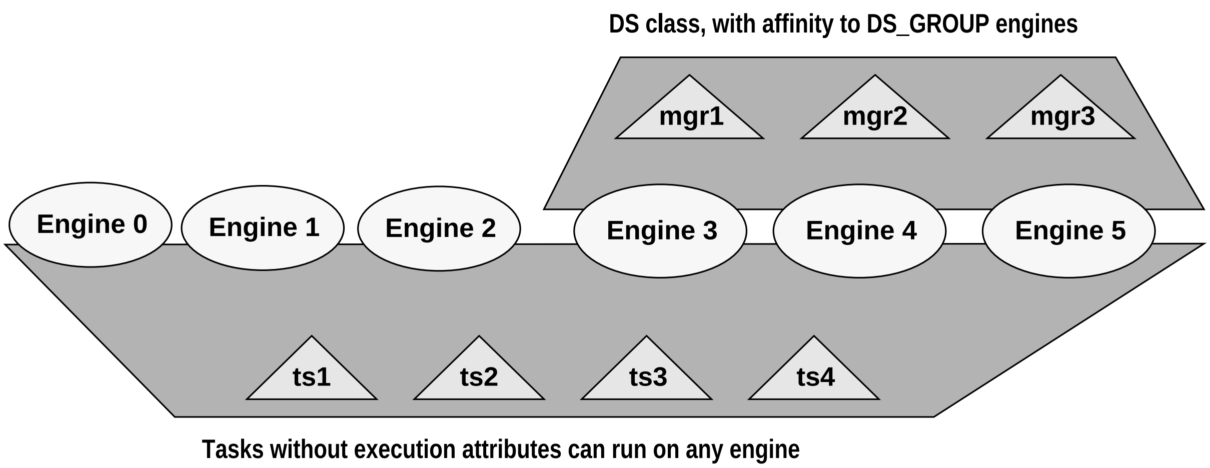


Figure 38-2: An example of engine affinity

How Execution Class Bindings Affect Scheduling

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

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

Execution Class Bindings

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

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

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

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

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

Figure 38-3 shows four execution objects running tasks. Several users are running the **order_entry** and **sales_report** applications. Two other logins are active, "mgr" (logged in once using the **sales_report** application, and twice using **isql**) and "cs3" (not using the affected applications).

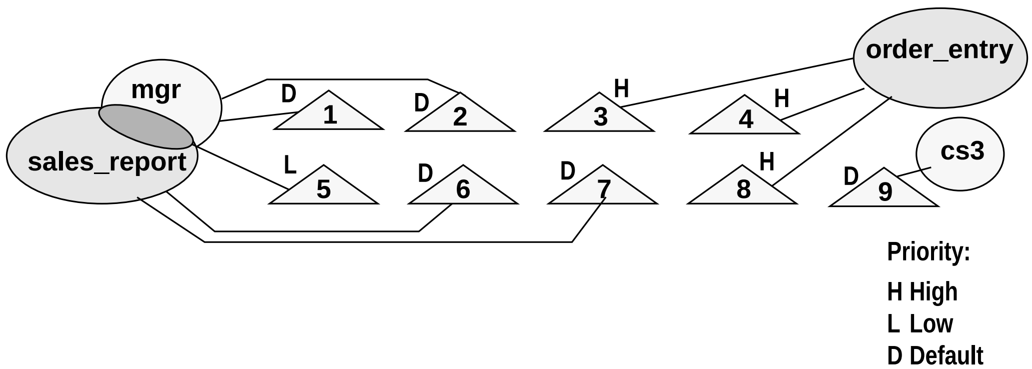


Figure 38-3: Execution objects and their tasks

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

How Engine Affinity Can Affect on Scheduling

Each execution class is associated with a different priority:

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

An engine looking for a task to run first looks in its own high-priority run queues, then in the high-priority global run queue. If there are no high-priority tasks, it checks for medium-priority tasks in its own run queue, then in the medium-priority global run queue, and finally for low-priority tasks. Figure 38-4 shows three engines, with tasks queued for each engine, and in the global run queue.

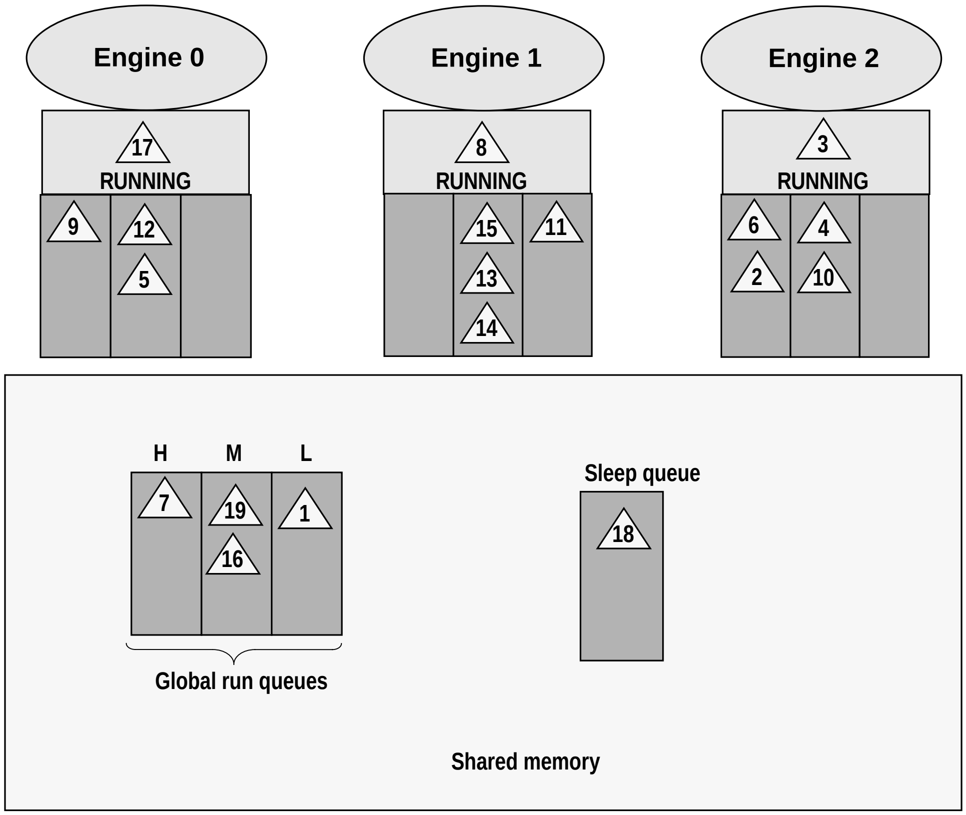


Figure 38-4: Per-engine and global run queues

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

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

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

Setting Attributes for a Session Only

If you need to change any attribute value temporarily for an active session, you can do so using **sp_setpsexe**. 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_setpsexe** 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. To clear attributes set for a session, use **sp_clearpsexe**.

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

sp_showexeclass displays the attribute values of an execution class or all execution classes. You can also use **sp_showpsexe** 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 can invoke an *EC1* stored procedure. Do both execution objects take *EC3* attributes, *EC1* attributes, or *EC2* attributes?

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

Multiple Execution Objects and ECs, 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 38-5.

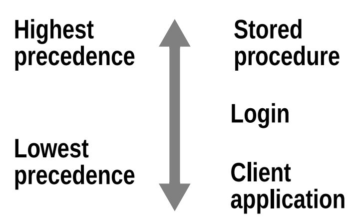


Figure 38-5: Precedence rule

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. Figure 38-6 shows how the precedence rule is applied.

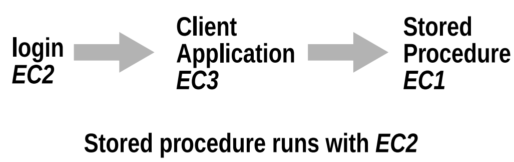


Figure 38-6: Use of the precedence rule

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

The Scope Rule

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

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

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

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

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

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

The execution object’s execution attributes apply to all of its interactions if the scope is NULL. When a client application has no scope, the execution attributes bound to it apply to any login that invokes the client application. When a login has no scope, the attributes apply to the login for any process that the login invokes.

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

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

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

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

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

Resolving a Precedence Conflict

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

- Execution objects not bound to a specific execution class are assigned the default values shown in the following table:

Entity Type	Attribute Name	Default Value
Client application	Execution class	*EC2*
Login	Execution class	*EC2*
Stored procedure	Execution class	*EC2*

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

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

Examples: Determining Precedence

Each row in Table 38-3 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 38-3: 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
EC1	*EC2*	*EC1* (*EC3*)	*EC2*	Medium	High (Medium)
EC1	*EC3*	*EC1* (*EC2*)	*EC3*	Low	High (Medium)
EC2	*EC1*	*EC2* (*EC3*)	*EC1*	High	High (High)
EC2	*EC3*	*EC1* (*EC2*)	*EC3*	Low	High (Medium)
EC3	*EC1*	*EC2* (*EC3*)	*EC1*	High	High (High)
EC3	*EC2*	*EC1* (*EC3*)	*EC2*	Medium	High (Medium)

To test your understanding of the rules of precedence and scope, cover the "Adaptive Server-Assigned Values" columns in Table 38-3, 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 38-7 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.

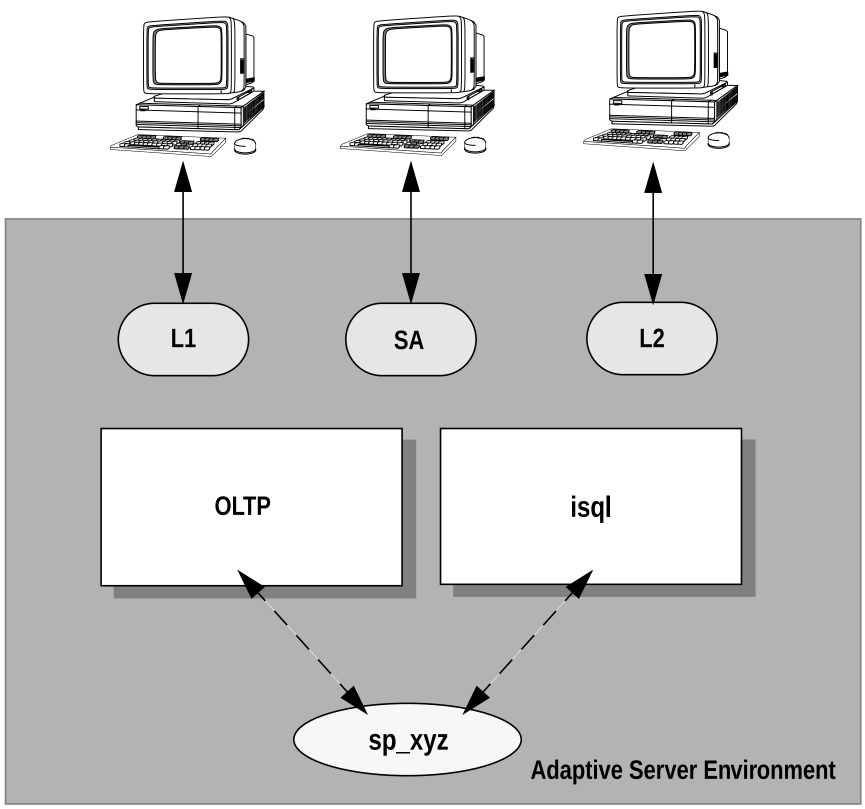


Figure 38-7: Conflict resolution

The rest of this section describes one way to implement the steps discussed in “Algorithm for Successfully Distributing Engine Resources” on page 38-24.

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 38-24 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 38-1 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 38-4: Example analysis of an Adaptive Server environment

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

Configuration

The System Administrator executes the following system procedures to assign execution classes (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 in to Adaptive Server as “L1” using **OLTP**.
   - Adaptive Server determines that **OLTP** is *EC1*.
   - “L1”does not have an execution class, so Adaptive Server assigns the default class *EC2*. “L1”gets the characteristics defined by *EC1* when it invokes **OLTP**.
   - If “L1”executes stored procedure **sp_xyz**, its priority remains unchanged while **sp_xyz** executes. During execution, “L1”has *EC1* attributes throughout.
2. A client logs in to Adaptive Server as “L1” using **isql**.
   - Because **isql** is *EC3*, and the “L1”execution class is undefined, “L1”executes with *EC3* characteristics. This means it runs at low priority and has affinity with the highest numbered engine (as long as there are multiple engines).
   - When “L1”executes **sp_xyz**, its priority is raised to high because the stored procedure is *EC1*.
3. A client logs in to Adaptive Server as “sa” using **isql**.
   - Adaptive Server determines the execution classes for both **isql** and the “sa”, using the precedence rule. Adaptive Server runs the System Administrator’s instance of **isql** with *EC1* attributes. When the System Administrator executes **sp_xyz**, the priority does not change.

4. A client logs in to Adaptive Server as “L2” using **isql**.
   - Because both the application and login are *EC3*, there is no conflict. “L2” executes **sp_xyz** at high priority.

Considerations for Engine Resource Distribution

Making execution class assignments indiscriminately does not usually yield what you expect. Certain conditions yield better performance for each execution object type. Table 38-5 indicates when assigning an execution precedence might be advantageous for each type of execution object.

Table 38-5: 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 nonpreferred 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 nonpreferred 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 only 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 38-8. A discussion of the steps follows the figure.

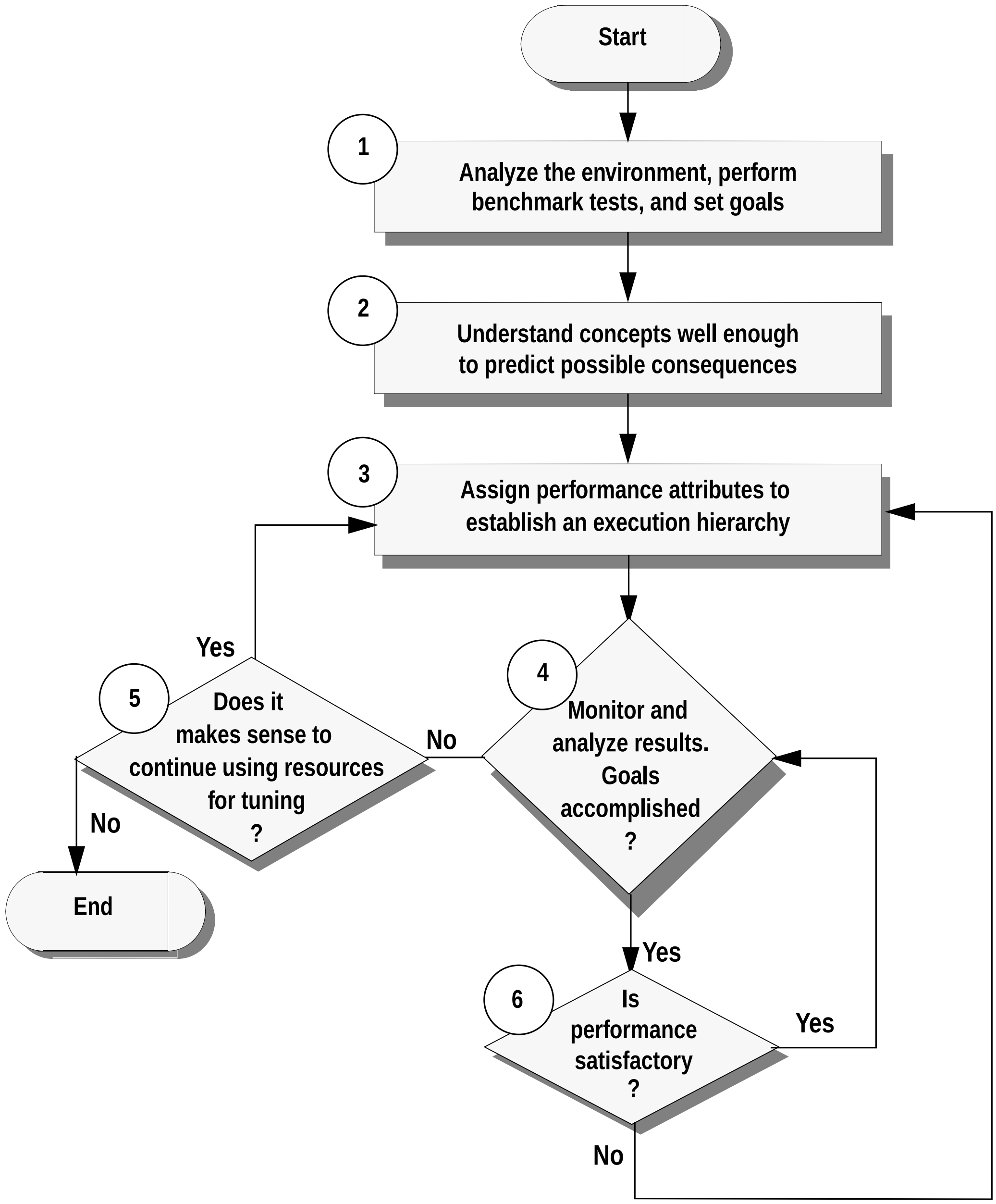


Figure 38-8: Process for assigning execution precedence

Algorithm Guidelines

1. Study the Adaptive Server environment. See “Environment Analysis and Planning” on page 38-27 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 38-3 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. After making execution precedence assignments. analyze the running Adaptive Server environment. See “Results Analysis and Tuning” on page 38-30 for details.
   - Run the benchmark tests you used in step 1 and compare the results.
   - If the results are not what you expect, take a closer look at the interactions between execution objects, as outlined in step 1.
   - Investigate dependencies that you might have missed.

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

Environment Analysis and Planning

This section elaborates on step 1 of “Algorithm for Successfully Distributing Engine Resources” on page 38-24.

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 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 will not normally use an entire time quantum, and will give up the CPU before waiting for I/O to complete. By giving preference to I/O-bound Adaptive Server tasks, Adaptive Server ensures that these tasks are runnable as soon as the I/O is finished.

By letting the I/O take place first, the CPU should be able to accommodate both types of applications and logins.

Example: Phase 2 – 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 39, “Monitoring Performance with sp_sysmon.” In particular, see “Application Management” on page 39-30.

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 38-27 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 39-30.
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 38-27.

   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.

39 Monitoring Performance with *sp_sysmon*

This chapter describes output from **sp_sysmon**, a system procedure that produces Adaptive Server performance data. it includes suggestions for interpreting its output and deducing possible implications. **sp_sysmon** output is most valuable when you have a good understanding of your Adaptive Server environment and its specific mix of applications. Otherwise, you may find that **sp_sysmon** output has little relevance.

This chapter contains the following sections:

Using *sp_sysmon*

When you invoke **sp_sysmon**, it clears all accumulated data from a set of counters that will be used during the sample interval to accumulate the results of user and system activity. At the end of the sample interval, the procedure reads the values in the counters, prints the report, and stops executing.

The flow diagram below shows the algorithm.

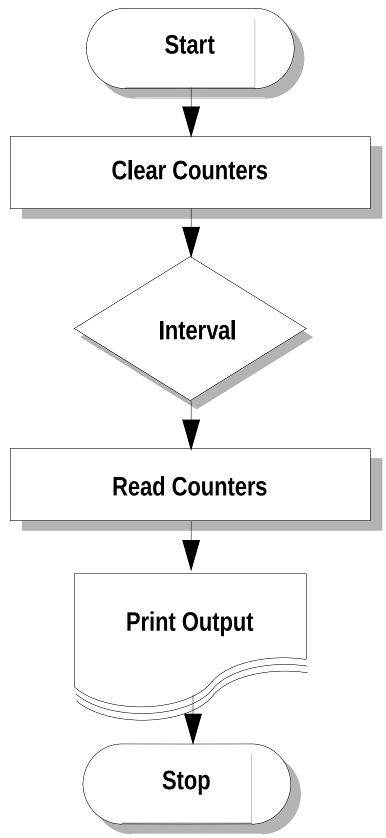


Figure 39-1: sp_sysmon execution algorithm

sp_sysmon contributes 5 to 7% 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.

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 or bottlenecks
- During stress tests to evaluate an 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 a 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 39-6.

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***

On systems with many CPUs and high activity, counters can overflow if the sample period is too long. 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 39-1 for the second parameter.

Table 39-1: 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**

Table 39-1: sp_sysmon report sections

Report Section	Parameter
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

If you specify the third parameter to **sp_sysmon**, the report includes detailed information by application or by application and login name. This parameter 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 and request any other section of the report.

The third 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 application and 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
```

See “Application Statistics per Application or per Application and Login” on page 39-34 for sample output.

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. Sometimes removing one performance bottleneck reveals another (see Figure 39-2). It is also possible that your tuning efforts might improve performance in one area, while actually causing performance degradation in another area.

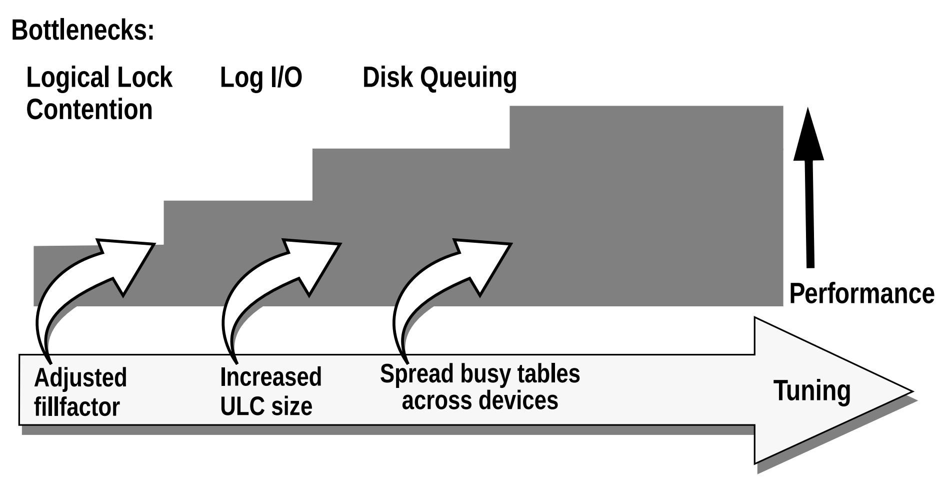


Figure 39-2: Eliminating one bottleneck reveals another

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. 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.

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% of 200 events is much more meaningful than 50% 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% of the time and another is working 2% of the time, a 49% 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/12.0/P/Sun_svr4/OS 5.6/1548/3
Server Name:            tinman
Run Date                Sep 20, 1999
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.

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 17-145 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% of a 10-minute sample interval, and “Engine Busy Utilization” was 90%, it means that Adaptive Server was busy for 7 minutes and 12 seconds and was idle for 48 seconds, as shown in Figure 39-3.

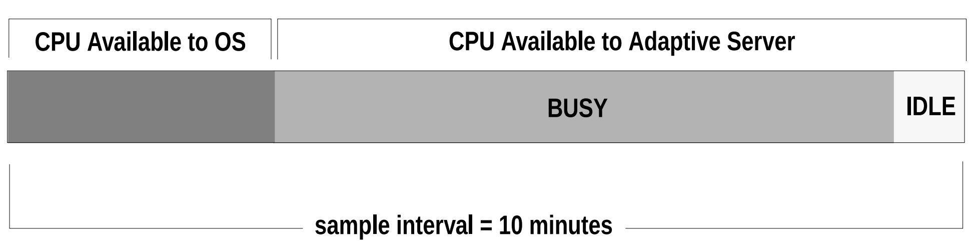


Figure 39-3: How Adaptive Server spends its available CPU time

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% range. If values are consistently very high (more than 90%), 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% 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 finds 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% 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, check for increased resource contention in other areas after adding each engine.

In an environment where Adaptive Server is serving a large number of users, performance is usually fairly evenly distributed across engines. However, when there are more engines than tasks, you may see some engines with a large percentage of utilization, and other engines may be idle. On a server with a single task running a query, for example, you may see output like this:

```
Engine Busy Utilization
    Engine 0                    97.2 %
    Engine 1                     0.0 %
    Engine 2                     0.0 %
    Engine 3                     0.0 %
    Engine 4                     0.0 %
    Engine 5                     0.0 %
  -----------          ---------------          ----------------
  Summary              Total    97.2 %          Average   16.2 %
```

In an SMP environment, tasks have soft affinity to engines. Without other activity (such as lock contention) that could cause this task to be placed in the global run cue, the task continues to run on the same 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 17-145 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 to the operating system 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 17-145 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 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 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 17-132 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 engines checked for 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 engine processes other tasks, if any, but also loops 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% of the checks were effective, and the other 90% 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 17-132 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 may 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 17-132 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 39-19.

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 multiple requests for worker processes. For example, a parallel query that requires a sort may make one request for accessing data and a second for parallel sort.

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 39-20.

“Requests Terminated” reports the number of times a request was terminated by user action, such as pressing Ctrl-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 for worker process** to 2048 bytes would require 50K of additional memory.

At start-up, 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 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 runtime, 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 runtime. Two possible causes are reported:

- "Due to WP Limit" indicates the number of times the number of worker processes for a cached query plan was adjusted due to a session-level limit set with **set parallel_degree** or **set scan_parallel_degree**. If "Due to WP Limit" is a nonzero value, look for applications that set session-level limits.
- "Due to No WPs" indicates the number of requests for which the number of worker processes was reduced due to lack of available worker processes. These queries may run in serial, or they may run in parallel with fewer worker processes than recommended by the optimizer. It could mean that queries are running with poorly-optimized plans.

  If "Due to No WPs" 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 or set session-level limits for some queries.

  Running **sp_showplan** on the *fid* (family ID) of a login using an adjusted plan shows only the cached plan, not the adjusted plan. If the login is running an adjusted plan, **sp_who** shows a different number of worker processes for the *fid* than the number indicated by **sp_showplan** results.

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 13-12 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 %
  Latch Contention              0.1        0.6       17       0.0 %
  Log Semaphore Contention     51.0        0.4     3084       5.7 %
  PLC Lock Contention           0.0        0.0        2       0.0 %
  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 %
  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 counts only connections that were started during the sample interval. Connections that were established before the interval started are not counted, although they may be active and using resources.

This 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. See “Transaction Profile” on page 39-39 for information about committed transactions.

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 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 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 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 39-62 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 "Scheduling Client Task Processing Time" on page 37-7 for more information.

A high number of voluntary yields indicates that there is little contention.

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 39-71 for more information about the cache-related parts of the **sp_sysmon** output.

System Disk Writes

"System 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 "System Disk Writes" seems high, check the value for page splits to see if the problem is caused by data page and index page splits. See "Page Splits" on page 39-55 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" reports how many times an I/O-intensive task was switched off an engine due to exceeding an I/O batch limit. Adaptive Server paces disk writes 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 due to contention for locks on tables, data pages, or data rows.

Investigate lock contention problems by checking the transaction detail and lock management sections of the report.

- See "Transaction Detail" on page 39-42 and "Lock Management" on page 39-62.
- Check to see if your queries are doing deferred and direct expensive updates, which can cause additional index locks. See "Updates" on page 39-44.
- Use **sp_object_stats** to report information on a per-object basis. See "Identifying Tables Where Concurrency Is a Problem" on page 28-11.

For additional help on locks and lock contention, check the following sources:

- "Types of Locks in Adaptive Server" on page 26-6 provides information about types of locks.
- "Reducing Lock Contention" on page 29-2 provides pointers on reducing lock contention.
- Chapter 9, "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 address locks. Adaptive Server acquires address locks on index pages of allpages-locked tables. Address lock contention blocks access to data pages.

Latch Contention

"Latch Contention" reports the number of times a task was switched out because it needed to wait for a latch. If your user tables use only allpages-locking, this latch contention is taking place either on a data-only-locked system table or on allocation pages. If your applications use data-only-locking, the contention reported here includes all waits for latches, including those on index pages and OAM pages as well as allocation pages.

Reducing Contention During Page Allocation

In SMP environments where inserts and expanding updates are extremely high, so that page allocations take place very frequently, contention for the allocation page latch can reduce performance. Normally, Adaptive Server allocates new pages for an object on an allocation unit that is already in use by the object and known to have free space. For each object, Adaptive Server tracks this allocation page number as a hint for any tasks that need to allocate a page for that object. When more than one task at a time needs to allocate a page on the same allocation unit, the second and subsequent tasks block on the latch on the allocation page.

You can specify a "greedy allocation" scheme, so that Adaptive Server keeps a list of eight allocation hints for page allocations for a table.

This command enables greedy allocation for the *salesdetail* table in database 6:

```
dbcc tune(des_greedyalloc, 6, salesdetail, "on")
```

To turn it off, use:

```
dbcc tune(des_greedyalloc, 6, salesdetail, "off")
```

The effect of **dbcc tune(des_greedyalloc)** are not persistent, so you need to reissue the commands after a reboot.

You should use this command only if all of the following are true:

- You have multiple engines. It is rarely useful with fewer than four engines.
- A large number of pages are being allocated for the object. You can use **sp_spaceused** or **optdiag** to track the number of pages.

- The latch contention counter shows contention.

Greedy allocation is more useful when tables are assigned to their own segments. If you enable greedy allocation for several tables on the same segment, the same allocation hint could be used for more than one table. Hints are unique for each table, but uniqueness is not enforced across all tables.

Greedy allocation is not allowed in the *master* and *tempdb* databases, and is not allowed on system tables.

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. If log semaphore contention is high, see "Transaction Management" on page 39-45.

Check disk queuing on the disk used by the transaction log. See "Disk I/O Management" on page 39-92.

Also see "Engine Busy Utilization" on page 39-12. 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.

PLC Lock Contention

"PLC Lock Contention" reports contention for a lock on a user log cache.

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, reported in "Transaction Profile" on page 39-39. 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 helps 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 number of tasks on 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 log 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 log write for the page is performed 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, see “Choosing the I/O Size for the Transaction Log” on page 32-25.

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 39-27.

If this value is high, review “Avg # Writes per Log Page” on page 39-51 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 a lightweight protection mechanism 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 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.

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 33-5 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 batch and was switched out waiting for the client to send the next 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 39-96 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 36-3 and "default network packet size" on page 17-90 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 packet to the client. The network model determines that there can be only one outstanding 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 data packet size is configurable, and different clients can request different packet sizes. For more information, see "Changing Network Packet Sizes" on page 36-3 and "default network packet size" on page 17-90 in the *System Administration Guide*.

If "Network Packet Sent" is a major cause of task switching, see "Network I/O Management" on page 39-96 for more information.

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. This section is useful if you use resource limits, or 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 38, "Distributing Engine Resources Between Tasks."

Requesting Detailed Application Information

If you request information about specific tasks using the third **sp_sysmon** parameter, **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 39-6 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. The first line identifies the application name (before the arrow) and the login name (after the arrow).

```
------------------------------------------------------------------------
  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 can help you determine whether there are any anomalies in resource utilization. If there are, you can investigate further using the detailed report.

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
- Whether tasks to which you have assigned low priority are getting starved for CPU time
- 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 39-11).

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 while the housekeeper sleeps, and changes 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.

If you are using logical process management and there are a high number of priority changes compared to steady state values, it may indicate that an application, or a user task related to that application, is changing priorities frequently. Check 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.

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 allotted for execution. Once a user task gains access to an engine, it is allowed to execute for a given period of time. If the task has not yielded the engine before the time 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. This happens when the task at the head of the run queue has affinity to an engine group and was bypassed in the queue by an engine that is not part of the engine group.

The value is affected by configuring engine groups and engine group bindings. A high number in this category might be acceptable if low priority tasks are bypassed for more critical tasks. It is possible that an engine group is bound so 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/0 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 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 39-92 and "Network I/O Management" on page 39-96.

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 18, "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.

Housekeeper Task Activity

The "Housekeeper Tasks Activity" section reports on housekeeper tasks. If the configuration parameter **housekeeper free write percent is set** to 0, the housekeeper task does not run. If **housekeeper free write percent** is 1 or greater, space reclamation can be enabled separately by setting **enable housekeeper GC** to 1, or disabled by setting it to 0.

Sample Output for Housekeeper Task Activity

```
Housekeeper Task Activity
-------------------------
                         per sec      per xact       count  % of total
                    ------------  ------------  ----------
Buffer Cache Washes
  Clean                     63.6           3.8       38163      96.7 %
  Dirty                      2.1           0.1        1283       3.3 %
                    ------------  ------------  ----------
Total Washes                65.7           3.9       39446

Garbage Collections          3.7           0.2        2230       n/a
Pages Processed in GC        0.0           0.0           1       n/a

Statistics Updates           3.7           0.2        2230       n/a
```

Buffer Cache Washes

This section reports:

- The number of buffers examined by the housekeeper
- The number that were found clean
- The number that were found dirty

The number of dirty buffers includes those already in I/O due to writes being started at the wash marker.

The "Recovery Management" section of **sp_sysmon** reports how many times the housekeeper task was able to write all dirty buffers for a database. See"Recovery Management" on page 39-88.

Garbage Collections

This section reports the number of times the housekeeper task checked to determine whether there were committed deletes that indicated that there was space that could be reclaimed on data pages.

“Pages Processed in GC” reports the number of pages where the housekeeper task succeeded in reclaiming unused space on the a page of a data-only-locked table.

Statistics Updates

“Statistics Updates” reports on the number of times the housekeeper task checked to see if statistics needed to be written.

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        n/a        n/a     4120       n/a
  (since beginning of sample)
```

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 17-109 of the *System Administration Guide*.

Transaction Profile

The "Transaction Profile" section reports on data modifications by type of command and table locking scheme.

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               16.5        n/a     9871     n/a

  Transaction Detail           per sec   per xact    count  % of total
  -------------------------  ---------  ---------  -------  ----------
    Inserts
      APL Heap Table             229.8       14.0   137900      98.6 %
      APL Clustered Table          2.5        0.2     1511       1.1 %
      Data Only Lock Table         0.9        0.1      512       0.4 %
  -------------------------  ---------  ---------  -------  ----------
    Total Rows Inserted          233.2       14.2   139923      91.5 %

    Updates
      APL Deferred                 0.5        0.0      287       2.3 %
      APL Direct In-place          0.0        0.0       15       0.1 %
      APL Direct Cheap             0.0        0.0        3       0.0 %
      APL Direct Expensive         0.0        0.0        0       0.0 %
      DOL Deferred                 0.4        0.0      255       2.1 %
      DOL Direct                  19.7        1.2    11802      95.5 %
  -------------------------  ---------  ---------  -------  ----------
```

```
  Total Rows Updated            20.6         1.3    12362        8.1 %

  Data Only Locked Updates
    DOL Replace                 19.6         1.2    11761       97.6 %
    DOL Shrink                   0.0         0.0        1        0.0 %
    DOL Cheap Expand             0.3         0.0      175        1.5 %
    DOL Expensive Expand         0.2         0.0      101        0.8 %
    DOL Expand & Forward         0.0         0.0       18        0.1 %
    DOL Fwd Row Returned         0.0         0.0        0        0.0 %
-------------------------  ---------  ---------  -------  ----------
  Total DOL Rows Updated        20.1         1.2    12056        7.9 %

  Deletes
    APL Deferred                 0.5         0.0      308       48.4 %
    APL Direct                   0.0         0.0        9        1.4 %
    DOL                          0.5         0.0      320       50.2 %
-------------------------  ---------  ---------  -------  ----------
  Total Rows Deleted             1.1         0.1      637        0.4 %
=========================  =========  =========  =======
  Total Rows Affected          254.9        15.5   152922
```

Transaction Summary

"Transaction Summary" reports committed transactions. "Committed Xacts" reports the number of transactions committed during the sample interval.

The count of transactions 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 39-4, both T1 and T2 are counted, but T3 is not.

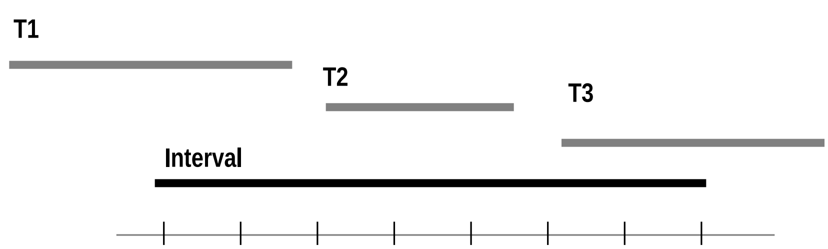


Figure 39-4: How transactions are counted

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.

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.

For the "Total Rows" for inserts, updates, and deletes, the "% of total" column reports the percentage of the transaction type as a percentage of all transactions.

See "Update Mode Messages" on page 17-7 for more information on deferred and direct inserts, updates, and deletes.

In the output for this section, APL indicates allpages-locked tables and DOL indicates data-only-locked tables.

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. It displays the number of inserts performed on:

- Allpages-locked heap tables
- Allpages-locked tables with clustered indexes
- Data-only locked tables

Insert statistics do 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.

APL Heap Tables

"APL Heap Tables" reports the number of row inserts that took place on allpages-locked 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
- 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 39-66. 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 33-16.

APL Clustered Table

“APL Clustered Table” reports the number of row inserts to allpages-locked 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 allpages-locked clustered tables can lead to page splitting. See “Row ID Updates from Clustered Split” and “Page Splits” on page 39-55.

Data Only Lock Table

“Data Only Lock Table” reports the number of inserts for all data-only-locked tables. The “% of total” column shows the percentage of inserts to data-only-locked tables as a percentage of all inserts.

Total Rows Inserted

“Total Rows Inserted” reports all row inserts to all 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 and Update Detail Sections

The “Updates” report has two sections, “Updates” and “Data Only Locked Updates.”

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:

- APL Deferred
- APL Direct In-place
- APL Direct Cheap
- APL Direct Expensive
- DOL Deferred
- DOL Direct

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.

For a description of update types, see "How Update Operations Are Performed" on page 6-30. If there is a high percentage of deferred updates, see "Optimizing Updates" on page 6-37.

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.

Data-Only-Locked Updates

This section reports more detail on updates to data-only-locked tables:

- DOL Replace – The update did not change the length of the row; some or all of the row was changed resulting in the same row length
- DOL Shrink – The update shortened the row, leaving noncontiguous empty space on the page to be collected during space reclamation.
- DOL Cheap Expand – The row grew in length; it was the last row on the page, so expanding the length of the row did not require moving other rows on the page.

- DOL Expensive Expand – The row grew in length and required movement of other rows on the page.
- DOL Expand and Forward – The row grew in length, and did not fit on the page. The row was forwarded to a new location.
- DOL Fwd Row Returned – The update affected a forwarded row; the row fit on the page at its original location and was returned to that page.

The total reported in “Total DOL Rows Updated” are not included in the “Total Rows Affected” sum at the end of the section, since the updates in this group are providing a different breakdown of the updates already reported in “DOL Deferred” and “DOL Direct.”

Deletes

“Deletes” reports the number of deferred and direct row deletes from allpages-locked tables. All deletes on data-only-locked tables are performed by marking the row as deleted on the page, so the categories “direct” and “deferred” do not apply. The “% of total” column reports the percentage of each type of delete as a percentage of the total number of 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 17-171 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. If the “% of total” value for “by Full ULC” is greater than 20%, 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.

By System Log Record and By Other

If either of these values is higher than approximately 20%, 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 39-49
- Contention for the log semaphore. (SMP only); see “Log Semaphore Requests” on page 39-50
- The number of transaction log writes; see “Transaction Log Writes” on page 39-51

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 use 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 39-47). If the number of times that logs were flushed due to a full ULC is more than 20%, consider increasing the **user log cache size** configuration parameter. See "user log cache size" on page 17-171 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%.

Some options for reducing log semaphore contention 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 39-46 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       20.4        1.2    12269       n/a
      # of NC Ndx Maint            5.9        0.4     3535       n/a
      Avg NC Ndx Maint / Op        n/a        n/a  0.28812       n/a

    Deletes Requiring Maint       20.4        1.2    12259       n/a
      # of NC Ndx Maint            5.9        0.4     3514       n/a
      Avg NC Ndx Maint / Op        n/a        n/a  0.28665       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

    Upd/Del DOL Req Maint          7.3        0.4     4351       n/a
      # of DOL Ndx Maint           4.7        0.3     2812       n/a
      Avg DOL Ndx Maint / Op       n/a        n/a  0.64629       n/a

  Page Splits                      0.3        0.0      207       n/a
    Retries                        0.0        0.0        1       0.5 %
    Deadlocks                      0.0        0.0        0       0.0 %
    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              717.1       43.6   430258      90.6 %
    DOL Ascending Scans           74.3        4.5    44551       9.4 %
    Descending Scans               0.1        0.0       85       0.0 %
    DOL Descending Scans           0.0        0.0        6       0.0 %
                             ---------  ---------  -------
    Total Scans                  791.5       48.1   474900
```

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 39-42, and “Page Splits” on page 39-55.
- Information on lock contention. See “Lock Detail” on page 39-66.
- Information on address lock contention. See “Address Lock Contention” on page 39-25 and “Address Locks” on page 39-67.

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 9, “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 on allpages-locked tables. 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.

- “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.

- "# of NC Ndx Maint" reports the number of nonclustered indexes that required maintenance as a result of insert and update operations.
- "Avg NC Ndx Maint / Op" reports the average number of nonclustered indexes per insert or update operation that required maintenance.

For data-only-locked tables, inserts are reported in "Ins / Upd Requiring Maint" and deletes and inserts are reported in "Upd / Del DOL Req Maint."

Deletes Requiring Maintenance

The data in this section gives information about how delete operations affected indexes on allpages-locked tables:

- "Deletes Requiring Maint" reports the number of delete operations that potentially required modification to one or more indexes. See "Deletes" on page 39-45.
- "# of NC Ndx Maint" reports the number of nonclustered indexes that required maintenance as a result of delete operations.
- "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 allpages-locked tables with clustered indexes. These splits require updating the nonclustered indexes for all of the rows that move to the new data page.

- "RID Upd from Clust Split" reports the total number of page splits that required maintenance of a nonclustered index.
- "# of NC Ndx Maint" reports the number of nonclustered rows that required maintenance as a result of row ID update operations.
- "Avg NC Ndx Maint / Op" reports the average number of nonclustered indexes entries that were updated for each page split.

Data-Only-Locked Updates and Deletes Requiring Maintenance

The data in this section gives information about how updates and deletes affected indexes on data-only-locked tables:

- "Upd / Del DOL Req Maint" reports the number of update and delete operations that potentially required modification to one or more indexes.
- "# of DOL Ndx Main" reports the number of indexes that required maintenance as a result of update or delete operations.
- "Avg DOL Ndx Maint / Op" reports the average number of indexes per update or delete operation that required maintenance.

Page Splits

"Page Splits" reports the number page splits for data pages, clustered index pages, or nonclustered index pages because there was not enough room for a new row.

When a data row is inserted into an allpages-locked 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 Space Management Properties for Indexes" on page 9-19 for more information about how to temporarily reduce page splits using **fillfactor**.

Reducing Page Splits for Ascending Key Inserts

If "Page Splits" is high and your application is inserting values into an allpages-locked 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 39-5.

Page 1007		
A	1	...
A	2	...
A	3	...
B	1	...

Page 1009		
B	2	...
C	1	...
C	2	...
C	3	...

Figure 39-5: 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 39-6.

Page 1007		
A	1	...
A	2	...

Page 1129		
A	3	...
A	4	...
B	1	...

Page 1009		
B	2	...
C	1	...
C	2	...
C	3	...

Figure 39-6: 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 39-7.

Page 1007		
A	1	...
A	2	...

Page 1129		
A	3	...
A	4	...

Page 1134		
A	5	...
A	6	...
B	1	...

Page 1009		
B	2	...
C	1	...
C	2	...
C	3	...

Figure 39-7: Another insert causes another page split

Adding “A,7” and “A,8” results in yet another page split, as shown in Figure 39-8.

Page 1007		
A	1	...
A	2	...

Page 1129		
A	3	...
A	4	...

Page 1134		
A	5	...
A	6	...

Page 1137		
A	7	...
A	8	...
B	1	...

Page 1009		
B	2	...
C	1	...
C	2	...
C	3	...

Figure 39-8: 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 39-5 on page 39-56, 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 39-9.

Page 1007		
A	1	...
A	2	...
A	3	...
A	4	...

Page 1129		
B	1	...

Page 1009		
B	2	...
C	1	...
C	2	...
C	3	...

Figure 39-9: First insert with ascending inserts mode

Inserting “A,5” causes a new page to be allocated, as shown in Figure 39-10.

Page 1007		
A	1	...
A	2	...
A	3	...
A	4	...

Page 1134		
A	5	...

Page 1129		
B	1	...

Page 1009		
B	2	...
C	1	...
C	2	...
C	3	...

Figure 39-10: Additional ascending insert causes a page allocation

Adding “A,6”, “A,7”, and “A,8” fills the new page, as shown in Figure 39-11.

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 39-11: 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 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.

By the time an index deadlock takes place, the transaction has already updated the data page and is holding data page locks so 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 index 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 17-69 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 “Reducing Index Maintenance with fillfactor” on page 31-1.

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

The “Index Scans” section reports forward and backward scans by lock scheme:

- “Ascending Scans” reports the number of forward scans on allpages-locked tables.
- “DOL Ascending Scans” reports the number of forward scans on data-only-locked tables.
- “Descending Scans” reports the number of backward scans on allpages-locked tables.
- “DOL Descending Scans” reports the number of backward scans on data-only-locked tables.

For more information on forward and backward scans, see “Query Costing for Queries Using order by” on page 6-16.

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 39-2.

Table 39-2: Action to take based on metadata cache usage statistics

***sp_sysmon* 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

Object and Index Spinlock Contention

These sections report on spinlock contention 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%, decrease the value of the corresponding parameter to lower the number of objects or indexes that are protected by a single spinlock.

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%, 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      2634.5         151.2     1580714       n/a
  Avg Lock Contention         2.4           0.1        1436       0.1 %
  Deadlock Percentage         0.0           0.0           1       0.0 %
  Lock Hashtable Lookups   8262.3         474.2     4957363       n/a
  Avg Hash Chain Length       n/a           n/a     0.01153       n/a

  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 Row
  Granted                        1.3        0.1      751       75.6 %
  Waited                         0.4        0.0      243       24.4 %
-------------------------  ---------  ---------  -------  ----------
Total EX-Row Requests            1.7        0.1      994        0.1 %

Update Row
  Granted                        0.2        0.0      155       62.0 %
  Waited                         0.3        0.0       95       38.0 %
-------------------------  ---------  ---------  -------  ----------
Total UP-Row Requests            0.4        0.0      250        0.0 %

Shared Row
  Granted                     1699.8      103.3  1019882      100.0 %
  Waited                         0.1        0.0       46        0.0 %
-------------------------  ---------  ---------  -------  ----------
Total SH-Row Requests         1699.9      103.3  1019928       59.7 %


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 Timeouts by Lock Type   per sec   per xact    count  % of total
-------------------------  ---------  ---------  -------  ----------
  Exclusive Table                 0.0        0.0        0       0.0 %
  Shared Table                    0.0        0.0        0       0.0 %
  Exclusive Intent                0.0        0.0        4      44.4 %
  Shared Intent                   0.0        0.0        0       0.0 %
  Exclusive Page                  0.0        0.0        0       0.0 %
  Update Page                     0.0        0.0        1      11.1 %
  Shared Page                     0.0        0.0        4      44.4 %
  Exclusive Row                   0.0        0.0        0       0.0 %
  Update Row                      0.0        0.0        0       0.0 %
  Shared Row                      0.0        0.0        0       0.0 %
  Exclusive Address               0.0        0.0        0       0.0 %
  Shared Address                  0.0        0.0        0       0.0 %
  Shared Next-Key                 0.0        0.0        0       0.0 %
-------------------------  ---------  ---------  -------
Total Lock Timeouts               0.0        0.0        9
```

"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" reports the total number of lock requests.
- "Avg Lock Contention" reports the average number of times there was lock contention as a percentage of the total number of lock requests.

  If the lock contention average is high, study the lock detail information below. See Chapter 29, "Locking Configuration and Tuning," for more information on tuning locking behavior.

- “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 39-67.
- “Lock Hashtable Lookups” reports the number of times the lock hash table was searched for a lock on a page, row, or table.
- “Avg Hash Chain Length” reports the average number of locks per hash bucket during the sample interval. You can configure the size of the lock hash table with the configuration parameter **lock hashtable size**. If the average number of locks per hash chain is more than four, consider increasing the size of the hash table. See “Configuring the Lock Hashtable” on page 29-8 for more information. Large inserts with bulk copy are an exception to this guideline. Lock hash chain lengths may be longer during large bulk copies.

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 Row
- Update Row
- Shared Row
- 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 or row 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. If lock contention is high, run **sp_object_stats** to help pinpoint the tables involved. See Chapter 28, "Identifying Tables Where Concurrency Is a Problem," for more information.

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 of allpages-locked tables. They can have serious impact, since a lock on an index page blocks access to all data pages pointed to by the index page.

Last Page Locks on Heaps

"Last Page Locks on Heaps" reports locking attempts on the last page of a partitioned or unpartitioned heap table. It only reports on allpages-locked tables.

This information can indicate whether there are tables in the system that would benefit from using data-only-locking or 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 33-16 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
- Exclusive Row
- Update Row
- Shared Row
- Shared Next-Key
- Exclusive Address
- Shared Address
- Others

"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, use one or both of the following methods:

- Use **sp_object_stats**. See Chapter 28, "Identifying Tables Where Concurrency Is a Problem," for more information.
- Enable printing of detailed deadlock information to the log. See "Printing Deadlock Information to the Error Log" on page 28-9.

For more information on deadlocks and coping with lock contention, see "Deadlocks and Concurrency" on page 28-5 and "Locking and Performance" on page 29-1.

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 28-5.

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 17-68 in the *System Administration Guide* for more information.

Average Deadlocks per Search

"Avg Deadlocks per Search" reports the average number of deadlocks found per search.

This category measures whether Adaptive Server is checking too frequently. If your applications rarely deadlock, 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 17-68 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.
- "Ex-Row to Ex-Table" – Exclusive row to exclusive table.
- "Sh-Row to Sh-Table – Shared row to shared table.
- "Sh-Next-Key to Sh-Table" – Shared next-key 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 26-16.

Lock Timeout Information

The "Lock Timeouts by Lock Type" section reports on the number of times a task was waiting for a lock and the transaction was rolled back due to a session-level or server-level lock timeout. The detail rows that show the lock types are printed only if lock timeouts occurred during the sample period. If no lock timeouts occurred, the "Total Lock Timeouts" row is displayed with all values equal to 0.

For more information on lock timeouts, see "Lock Timeouts" on page 27-14.

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 39-12 shows how these caching features relate to disk I/O and the data caches.

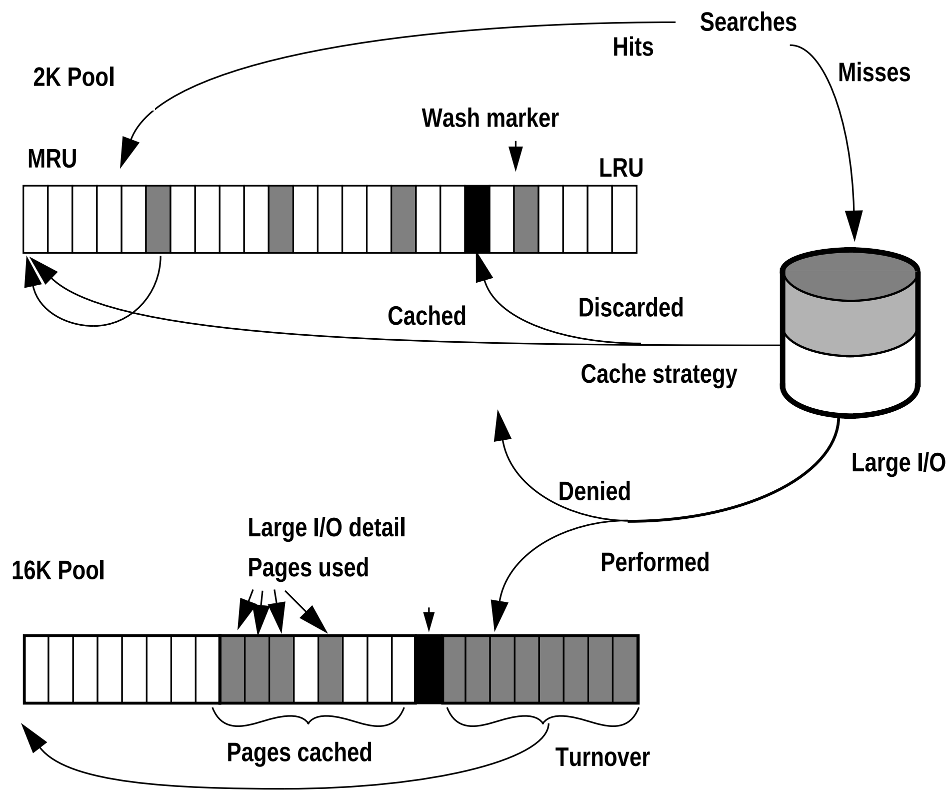


Figure 39-12: Cache management categories

You can use **sp_cacheconfig** and **sp_helpcache** output to help 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 15, "Configuring Data Caches," in the *System Administration Guide* for information on how to use these system procedures. See "Configuring the Data Cache to Improve Performance" on page 32-12 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. **sp_sysmon** reports a separate block of data for each cache. These blocks are identified by the cache name. The sample output shown here includes only the default data 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. See "Cache Management By Cache" on page 39-80.

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" 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" 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" 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" 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" 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" reports the total number of buffers placed at the head of the MRU/LRU chain in all caches.
- "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" 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" 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" 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. Use **optdiag** to check the value for "Large I/O Efficiency" for each table and index.

- "Pages by Lrg I/O Cached" reports the number of pages brought into all caches by all 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" reports the total number of pages that were used after being brought into cache by 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 39-92.

“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 39-93.
  - 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 39-96 for more information.

  If the problem is due to a shortage of 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 that were 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.

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.

Cache 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 10%, consider using named caches or adding cache partitions.

See "Configuring the Data Cache to Improve Performance" on page 32-12 for information on adding caches, and "Reducing Spinlock Contention with Cache Partitions" on page 32-18.

Utilization

"Utilization" reports the percentage of searches using this cache as a percentage of searches across all caches.You can compare this value for each cache to determine if there are caches that are over- or under-utilized. If you decide that a cache is not well utilized, you can:

- Change the cache bindings to balance utilization. For more information, see "Caches and Object Bindings" on page 3-20 and "Binding Objects to Caches" on page 15-15 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 15-24 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%. 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 area is too big. 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 "Buffer Wash Behavior" on page 39-84 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 "Cache Strategy" on page 39-84 for more information, and "Specifying the Cache Strategy" on page 20-10 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 15-20 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 "Cache Turnover" on page 39-75.

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% 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% of the buffers in the pool.
- If the cache is partitioned, reduce the number of partitions.
- 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" and "Buffers Washed Dirty" in the section "Buffer Wash Behavior" on page 39-84. 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 39-92 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 17-126 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:

- "Buffers Passed Clean" reports the number of buffers that were clean when they passed the wash marker. 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. "% 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" reports the number of times that I/O was already active on a buffer when it entered the wash area. 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. "% 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.
- "Buffers Washed Dirty" reports the number of times that a buffer entered the wash area dirty and not already in I/O. 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. "% of total" reports the percentage of buffers washed dirty as a percentage of the total number of buffers that entered the wash area.

If no buffers pass the wash marker during the sample interval, **sp_sysmon** prints:

```
Statistics Not Available - No Buffers Entered Wash
Section Yet!
```

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" 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" 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 e 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 20-10 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.

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. Re-run 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.

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 by Lrg I/O Cached" prints the total number of pages read into the cache.
- "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 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 found and 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 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 data caches and writes out all changed data pages.

When Adaptive Server has no user tasks to process, a housekeeper task 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 performed 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 17-24 and "Synchronizing a Database and Its Log: Checkpoints" on page 26-2 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 performed by the housekeeper task. The housekeeper performs checkpoints only when it has cleared all dirty pages from all configured caches.

You can use the **housekeeper free write percent** parameter to configure the maximum percentage by which the housekeeper task can increase database writes. See "housekeeper free write percent" on page 17-126 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 average time that normal checkpoints lasted.

Average Time per Free Checkpoint

"Avg Time per Free Chkpt" reports the average time 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 go to the same device, it moves on 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 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
```

```
d_master
master                        per sec   per xact    count  % of total
-------------------------  ---------  ---------  -------  ----------
  Reads
     APF                        56.6        0.5     3423      46.9 %
     Non-APF
  Writes                        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 17-40 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 17-99 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 17-99 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 17-40 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 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%.

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 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 33-11.

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. On SMP systems, multiple engines can try 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 network I/O. 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 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 packet size, Adaptive Server waits 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 36-3 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

Numerics

A

B

C

D

F

G

H

I

M

N

O

P

S

T

U

V

W