



Performance and Tuning Series:
Locking and Concurrency Control

Adaptive Server® Enterprise

15.0.2

DOCUMENT ID: DC00938-01-1502-01

LAST REVISED: October 2008

Copyright © 2008 by Sybase, Inc. All rights reserved.

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

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

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

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

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

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

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

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

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

Contents

| | |
|-----------------------|-----|
| About This Book | vii |
|-----------------------|-----|

| | | |
|------------------|---|----------|
| CHAPTER 1 | Introduction to Locking | 1 |
| | How locking affects performance | 1 |
| | Locking and data consistency | 2 |
| | Granularity of locks and locking schemes | 3 |
| | Allpages locking | 4 |
| | Datapages locking | 6 |
| | Datarows locking | 7 |
| | Types of locks in Adaptive Server | 7 |
| | Page and row locks | 8 |
| | Table locks | 10 |
| | Demand locks | 13 |
| | Row-locked system tables | 16 |
| | Range locking for serializable reads | 16 |
| | Latches | 17 |
| | Lock compatibility and lock sufficiency | 18 |
| | How isolation levels affect locking | 19 |
| | Isolation level 0, read uncommitted | 20 |
| | Isolation level 1, read committed | 21 |
| | Isolation level 2, repeatable read | 23 |
| | Isolation level 3, serializable reads | 23 |
| | Adaptive Server default isolation level | 25 |
| | Lock types and duration during query processing | 26 |
| | Lock types during create index commands | 29 |
| | Locking for select queries at isolation level 1 | 29 |
| | Table scans and isolation levels 2 and 3 | 30 |
| | When update locks are not required | 30 |
| | Locking during or processing | 31 |
| | Skipping uncommitted inserts during selects | 32 |
| | Using alternative predicates to skip nonqualifying rows | 33 |
| | Pseudocolumn-level locking | 34 |
| | Select queries that do not reference the updated column | 35 |
| | Qualifying old and new values for uncommitted updates | 36 |

| | | |
|------------------|---|-----------|
| | Reducing contention | 37 |
| CHAPTER 2 | Locking Configuration and Tuning | 39 |
| | Locking and performance..... | 39 |
| | Using sp_sysmon and sp_object_stats | 40 |
| | Reducing lock contention | 40 |
| | Additional locking guidelines | 43 |
| | Configuring locks and lock promotion thresholds..... | 44 |
| | Configuring the Adaptive Server lock limit..... | 44 |
| | Configuring the lock hashtable (lock manager) | 46 |
| | Setting lock promotion thresholds | 47 |
| | Choosing the locking scheme for a table | 53 |
| | Analyzing existing applications..... | 53 |
| | Choosing a locking scheme based on contention statistics | 54 |
| | Monitoring and managing tables after conversion..... | 56 |
| | Applications not likely to benefit from data-only locking | 56 |
| | Optimistic index locking..... | 57 |
| | Using optimistic index locking | 58 |
| | Cautions and issues | 58 |
| CHAPTER 3 | Locking Reports | 61 |
| | Locking tools | 61 |
| | Getting information about blocked processes | 61 |
| | Viewing locks with sp_lock | 63 |
| | Viewing locks with sp_familylock..... | 66 |
| | Intrafamily blocking during network buffer merges..... | 66 |
| | Deadlocks and concurrency | 67 |
| | Server-side versus application-side deadlocks | 67 |
| | Server task deadlocks | 67 |
| | Deadlocks and parallel queries | 69 |
| | Printing deadlock information to the error log..... | 70 |
| | Avoiding deadlocks | 71 |
| | Identifying tables where concurrency is a problem | 73 |
| | Lock management reporting | 74 |
| CHAPTER 4 | Using Locking Commands | 77 |
| | Specifying the locking scheme for a table..... | 77 |
| | Specifying a server-wide locking scheme | 77 |
| | Specifying a locking scheme with create table..... | 78 |
| | Changing a locking scheme with alter table | 78 |
| | Before and after changing locking schemes | 79 |
| | Expense of switching to or from allpages locking..... | 80 |

| | |
|--|----|
| Sort performance during alter table | 81 |
| Specifying a locking scheme with select into | 81 |
| Controlling isolation levels | 82 |
| Setting isolation levels for a session | 82 |
| Syntax for query-level and table-level locking options | 83 |
| Using holdlock, noholdlock, or shared | 83 |
| Using the at isolation clause | 84 |
| Making locks more restrictive | 85 |
| Making locks less restrictive | 86 |
| Readpast locking | 86 |
| Cursors and locking | 87 |
| Using the shared keyword | 88 |
| Additional locking commands | 89 |
| lock table | 89 |
| Lock timeouts | 90 |

CHAPTER 5

| | |
|--|-----------|
| Indexes | 91 |
| Types of indexes | 92 |
| Index pages | 93 |
| Index size | 94 |
| Indexes and partitions | 95 |
| Local indexes on partitioned tables | 95 |
| Global indexes on partitioned tables | 95 |
| Local versus global indexes | 96 |
| Unsupported partition index types | 96 |
| Clustered indexes on allpages-locked tables | 96 |
| Clustered indexes and select operations | 97 |
| Clustered indexes and insert operations | 98 |
| Page splitting on full data pages | 99 |
| Page splitting on index pages | 101 |
| Performance impacts of page splitting | 101 |
| Overflow pages | 102 |
| Clustered indexes and delete operations | 103 |
| Nonclustered indexes | 105 |
| Leaf pages revisited | 105 |
| Nonclustered index structure | 106 |
| Nonclustered indexes and select operations | 107 |
| Nonclustered index performance | 108 |
| Nonclustered indexes and insert operations | 109 |
| Nonclustered indexes and delete operations | 110 |
| Clustered indexes on data-only-locked tables | 111 |
| Index covering | 111 |
| Covering matching index scans | 112 |
| Covering nonmatching index scans | 113 |

| | |
|--|------------|
| Indexes and caching..... | 114 |
| Using separate caches for data and index pages..... | 115 |
| Index trips through the cache | 115 |
| CHAPTER 6 | |
| Indexing for Concurrency Control..... | 117 |
| How indexes affect performance | 117 |
| Detecting indexing problems | 119 |
| Symptoms of poor indexing | 119 |
| Fixing corrupted indexes | 122 |
| Index limits and requirements..... | 125 |
| Choosing indexes | 125 |
| Index keys and logical keys | 127 |
| Guidelines for clustered indexes..... | 127 |
| Choosing clustered indexes..... | 128 |
| Candidates for nonclustered indexes | 128 |
| Choosing function-based indexes..... | 129 |
| Index selection..... | 129 |
| Other indexing guidelines | 132 |
| Choosing nonclustered indexes..... | 133 |
| Choosing composite indexes..... | 134 |
| Key order and performance in composite indexes | 134 |
| Advantages and disadvantages of composite indexes..... | 136 |
| Using online reorg rebuild for data-only-locked indexes..... | 137 |
| Techniques for choosing indexes | 137 |
| Choosing an index for a range query..... | 137 |
| Adding a point query with different indexing requirements | 138 |
| Index and statistics maintenance | 140 |
| Dropping indexes that hurt performance | 140 |
| Choosing space management properties for indexes | 140 |
| Additional indexing tips..... | 141 |
| Creating artificial columns..... | 141 |
| Keeping index entries short and avoiding overhead..... | 141 |
| Dropping and rebuilding indexes | 142 |
| Configuring enough sort buffers | 142 |
| Creating the clustered index first | 142 |
| Configure large buffer pools | 143 |
| Asynchronous log service..... | 143 |
| Understanding the user log cache (ULC) architecture..... | 144 |
| When to use ALS..... | 145 |
| Using ALS..... | 146 |
| Index..... | 147 |

About This Book

| | |
|-----------------------------|---|
| Audience | This manual is intended for database administrators, database designers, developers and system administrators. |
| How to use this book | <ul style="list-style-type: none">• Chapter 1, “Introduction to Locking” – describes the types of locks that Adaptive Server uses and what types of locks are acquired during query processing.• Chapter 2, “Locking Configuration and Tuning” – describes the impact of locking on performance and describes the tools to analyze locking problems and configure locking.• Chapter 3, “Locking Reports” – describes the system procedures that report on locks and lock contention• Chapter 4, “Using Locking Commands” – describes the commands that set locking schemes for tables and control isolation levels and other locking behavior during query processing• Chapter 5, “Indexes” – provides guidelines and examples for choosing indexes• Chapter 6, “Indexing for Concurrency Control” – 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. |
| Related documents | <p>The Adaptive Server[®] Enterprise documentation set consists of the following:</p> <ul style="list-style-type: none">• 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 <i>Installation Guide</i> 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 15.0, the system changes added to support those features, and changes that may affect your existing applications.
 - *ASE Replicator User's Guide* – describes how to use the Adaptive Server Replicator feature of Adaptive Server to implement basic replication from a primary server to one or more remote Adaptive Servers.
 - *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.
 - *The Configuration Guide* for your platform – provides instructions for performing specific configuration tasks for Adaptive Server.
 - *Enhanced Full-Text Search Specialty Data Store User's Guide* – describes how to use the Full-Text Search feature with Verity to search Adaptive Server Enterprise data.
 - *Glossary* – defines technical terms used in the Adaptive Server documentation.
 - *Historical Server User's Guide* – describes how to use Historical Server to obtain performance information for SQL Server® and Adaptive Server.
 - *Java in Adaptive Server Enterprise* – describes how to install and use Java classes as datatypes, functions, and stored procedures in the Adaptive Server database.
 - *Job Scheduler User's Guide* – provides instructions on how to install and configure, and create and schedule jobs on a local or remote Adaptive Server using the command line or a graphical user interface (GUI).
 - *Messaging Service User's Guide* – describes how to use Real Time Messaging Services to integrate TIBCO Java Message Service and IBM WebSphere MQ messaging services with all Adaptive Server database applications.
 - *Monitor Client Library Programmer's Guide* – describes how to write Monitor Client Library applications that access Adaptive Server performance data.
 - *Monitor Server User's Guide* – describes how to use Monitor Server to obtain performance statistics from SQL Server and Adaptive Server.
 - *Performance and Tuning Series* – a series of books that explain how to tune Adaptive Server for maximum performance:

- *Basics* – the basics for understanding and investigating performance questions in Adaptive Server.
- *Locking and Concurrency Control* – describes how the various locking schemas can be used for improving performance in Adaptive Server, and how to select indexes to minimize concurrency.
- *Query Processing and Abstract Plans* – describes how the optimizer processes queries and how abstract plans can be used to change some of the optimizer plans.
- *Physical Database Tuning* – describes how to manage physical data placement, space allocated for data, and the temporary databases.
- *Monitoring Adaptive Server with sp_sysmon* – describes how to monitor Adaptive Server’s performance with sp_sysmon.
- *Improving Performance with Statistical Analysis* – describes how Adaptive Server stores and displays statistics, and how to use the set statistics command to analyze server statistics.
- *Using the Monitoring Tables* – describes how to query Adaptive Server’s monitoring tables for statistical and diagnostic information.
- *Quick Reference Guide* – provides a comprehensive listing of the names and syntax for commands, functions, system procedures, extended system procedures, data types, and utilities in a pocket-sized book (regular size when viewed in PDF format).
- *Reference Manual* – is a series of four books that contains the following detailed Transact-SQL information:
 - *Building Blocks* – Transact-SQL datatypes, functions, global variables, expressions, identifiers and wildcards, and reserved words.
 - *Commands* – Transact-SQL commands.
 - *Procedures* – Transact-SQL system procedures, catalog stored procedures, system extended stored procedures, and dbcc stored procedures.
 - *Tables* – Transact-SQL system tables and dbcc tables.
- *System Administration Guide*

-
- *Volume 1* – provides an introduction to the basics of system administration, including a description of configuration parameters, resource issues, character sets, sort orders, and diagnosing system problems. The second part of this book is an in-depth description of security administration.
 - *Volume 2* – includes instructions and guidelines for managing physical resources, mirroring devices, configuring memory and data caches, managing multiprocessor servers and user databases, mounting and unmounting databases, creating and using segments, using the `reorg` command, and checking database consistency. The second half of this book describes how to back up and restore system and user databases.
 - *System Tables Diagram* – illustrates system tables and their entity relationships in a poster format. Full-size available only in print version; a compact version is available in PDF format.
 - *Transact-SQL User's Guide* – documents Transact-SQL, the Sybase 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.
 - *User Guide for Encrypted Columns* – describes how configure and use encrypted columns with Adaptive Server
 - *Using Adaptive Server Distributed Transaction Management Features* – explains how to configure, use, and troubleshoot Adaptive Server DTM features in distributed transaction processing environments.
 - *Using Sybase Failover in a High Availability System* – provides instructions for using Sybase Failover to configure an Adaptive Server as a companion server in a high availability system.
 - *Unified Agent and Agent Management Console* – describes the Unified Agent, which provides runtime services to manage, monitor and control distributed Sybase resources.
 - *Utility Guide* – documents the Adaptive Server utility programs, such as `isql` and `bcp`, which are executed at the operating system level.
 - *Web Services User's Guide* – explains how to configure, use, and troubleshoot Web Services for Adaptive Server.

- *XA Interface Integration Guide for CICS, Encina, and TUXEDO* – provides instructions for using the Sybase DTM XA interface with X/Open XA transaction managers.
- *XML Services in Adaptive Server Enterprise* – describes the Sybase native XML processor and the Sybase Java-based XML support, introduces XML in the database, and documents the query and mapping functions that comprise XML Services.

Other sources of information

Use the Sybase Getting Started CD, the SyBooks CD, and the Sybase Product Manuals Web site to learn more about your product:

- The Getting Started CD contains release bulletins and installation guides in PDF format, and may also contain other documents or updated information not included on the SyBooks CD. It is included with your software. To read or print documents on the Getting Started CD, you need Adobe Acrobat Reader, which you can download at no charge from the Adobe Web site using a link provided on the CD.
- The SyBooks CD contains product manuals and is included with your software. The Eclipse-based SyBooks browser allows you to access the manuals in an easy-to-use, HTML-based format.

Some documentation may be provided in PDF format, which you can access through the PDF directory on the SyBooks CD. To read or print the PDF files, you need Adobe Acrobat Reader.

Refer to the *SyBooks Installation Guide* on the Getting Started CD, or the *README.txt* file on the SyBooks CD for instructions on installing and starting SyBooks.

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

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

Sybase certifications on the Web

Technical documentation at the Sybase Web site is updated frequently.

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

- 1 Point your Web browser to Technical Documents at <http://www.sybase.com/support/techdocs/>.
- 2 Click Certification Report.

-
- 3 In the Certification Report filter select a product, platform, and timeframe and then click Go.
 - 4 Click a Certification Report title to display the report.

v **Finding the latest information on component certifications**

- 1 Point your Web browser to Availability and Certification Reports at <http://certification.sybase.com/>.
- 2 Either select the product family and product under Search by Base Product; or select the platform and product under Search by Platform.
- 3 Select Search to display the availability and certification report for the selection.

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

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

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

Sybase EBFs and software maintenance

v **Finding the latest information on EBFs and software maintenance**

- 1 Point your Web browser to the Sybase Support Page at <http://www.sybase.com/support>.
- 2 Select EBFs/Maintenance. If prompted, enter your MySybase user name and password.
- 3 Select a product.
- 4 Specify a time frame and click Go. A list of EBF/Maintenance releases is displayed.

Padlock icons indicate that you do not have download authorization for certain EBF/Maintenance releases because you are not registered as a Technical Support Contact. If you have not registered, but have valid information provided by your Sybase representative or through your support contract, click Edit Roles to add the “Technical Support Contact” role to your MySybase profile.

- 5 Click the Info icon to display the EBF/Maintenance report, or click the product description to download the software.

Conventions

The following sections describe conventions used in this manual.

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 most 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. Complex commands are formatted using modified Backus Naur Form (BNF) notation.

Table 1 shows the conventions for syntax statements that appear in this manual:

Table 1: Font and syntax conventions for this manual

| Element | Example |
|---|--|
| Command names, procedure names, utility names, and other keywords display in sans serif font. | <code>select</code> <code>sp_configure</code> |
| Database names and datatypes are in sans serif font. | <code>master database</code> |
| Book names, file names, variables, and path names are in italics. | <i>System Administration Guide</i> <i>sql.ini</i> file <i>column_name</i> <i>\$SYBASE/ASE</i> directory |
| Variables—or words that stand for values that you fill in—when they are part of a query or statement, are in italics in Courier font. | <code>select column_name</code> <code>from table_name</code> <code>where search_conditions</code> |
| Type parentheses as part of the command. | <code>compute row_aggregate (column_name)</code> |
| Double colon, equals sign indicates that the syntax is written in BNF notation. Do not type this symbol. Indicates “is defined as”. | <code>::=</code> |
| Curly braces mean that you must choose at least one of the enclosed options. Do not type the braces. | <code>{cash, check, credit}</code> |
| Brackets mean that to choose one or more of the enclosed options is optional. Do not type the brackets. | <code>[cash check credit]</code> |
| The comma means you may choose as many of the options shown as you want. Separate your choices with commas as part of the command. | <code>cash, check, credit</code> |
| The pipe or vertical bar () means you may select only one of the options shown. | <code>cash check credit</code> |

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

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

```
sp_dropdevice [device_name]
```

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. *Italic font* shows user-supplied words.

- Examples showing the use of Transact-SQL commands are printed like this:

```
select * from publishers
```

- Examples of output from the computer appear as follows:

| pub_id | pub_name | city | state |
|--------|----------------------|------------|-------|
| ----- | ----- | ----- | ----- |
| 0736 | New Age Books | Boston | MA |
| 0877 | Binnet & Hardley | Washington | DC |
| 1389 | Algodata Infosystems | Berkeley | CA |

(3 rows affected)

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.

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. For more information, see the *System Administration Guide*.

Accessibility features

This document is available in an HTML version that is specialized for accessibility. You can navigate the HTML with an adaptive technology such as a screen reader, or view it with a screen enlarger.

Adaptive Server HTML documentation has been tested for compliance with U.S. government Section 508 Accessibility requirements. Documents that comply with Section 508 generally also meet non-U.S. accessibility guidelines, such as the World Wide Web Consortium (W3C) guidelines for Web sites.

Note You might need to configure your accessibility tool for optimal use. Some screen readers pronounce text based on its case; for example, they pronounce ALL UPPERCASE TEXT as initials, and MixedCase Text as words. You might find it helpful to configure your tool to announce syntax conventions. Consult the documentation for your tool.

For information about how Sybase supports accessibility, see Sybase Accessibility at <http://www.sybase.com/accessibility>. The Sybase Accessibility site includes links to information on Section 508 and W3C standards.

If you need help

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

Introduction to Locking

This chapter discusses basic locking concepts and the locking schemes and types of locks used in Adaptive Server.

| Topic | Page |
|---|------|
| How locking affects performance | 1 |
| Locking and data consistency | 2 |
| Granularity of locks and locking schemes | 3 |
| Types of locks in Adaptive Server | 7 |
| Lock compatibility and lock sufficiency | 18 |
| How isolation levels affect locking | 19 |
| Lock types and duration during query processing | 26 |
| Pseudocolumn-level locking | 34 |
| Reducing contention | 37 |

How locking affects performance

Adaptive Server protects the tables, data pages, or data rows 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 necessary in a multiuser environment, since several users may be working with the same data at the same time.

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

A more serious impact on performance arises from deadlocks. A **deadlock** occurs when two user processes each have a lock on a page, row, or table and each process wants to acquire a lock on the page, row, or table held by the other. 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 to reduce lock contention and avoid or minimize deadlocks.

Locking and data consistency

Data consistency 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 Table 1-1, 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.

Table 1-1: Consistency levels in transactions

| 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 |
| update account set balance = balance + 100 where acct_number = 45 | T1 updates balance of the other account by adding the \$100. | commit transaction |
| commit transaction | T1 ends. | |

If 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 is different by \$100. While such behavior may be acceptable in some situations, most database transactions must 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 is released when T1 is completed.

The alternative, returning data from uncommitted transactions, is known as a **dirty read**. If results do not need to be exact, T2 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 should know how and when to use transactions to preserve data consistency 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 an 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:

- A table-level lock
- A lock for each page that contains a required row
- A lock on each row

Less overall work is required when a table-level lock is used, but large-scale locks can degrade performance by making other users wait until locks are released. Decreasing lock granularity makes more data accessible to other users. Finer granularity locks can 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 data pages and index pages

- Datapages locking, which locks only data pages
- Datarows locking, which locks only data rows

For each locking scheme, Adaptive Server can lock an entire table, for queries that acquire many page or row locks, or can lock only the affected pages or rows.

Note The terms “data-only-locking” and “data-only-locked table” refer to both the datapages and datarows locking schemes, and are typically referred to as “DOL” tables. Allpages-locked tables are known as “APL” tables.

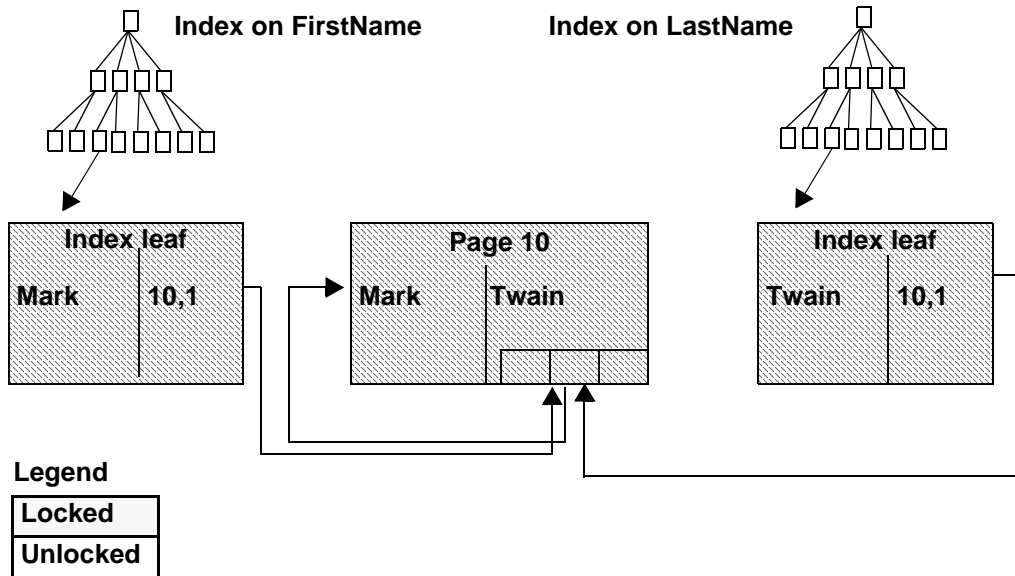
Allpages locking

Allpages locking locks 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 1-1 shows the locks acquired on data pages and indexes while a new row is being inserted into an allpages-locked table.

Figure 1-1: Locks held during allpages locking

insert authors values ("Mark", "Twain")



In many cases, 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.

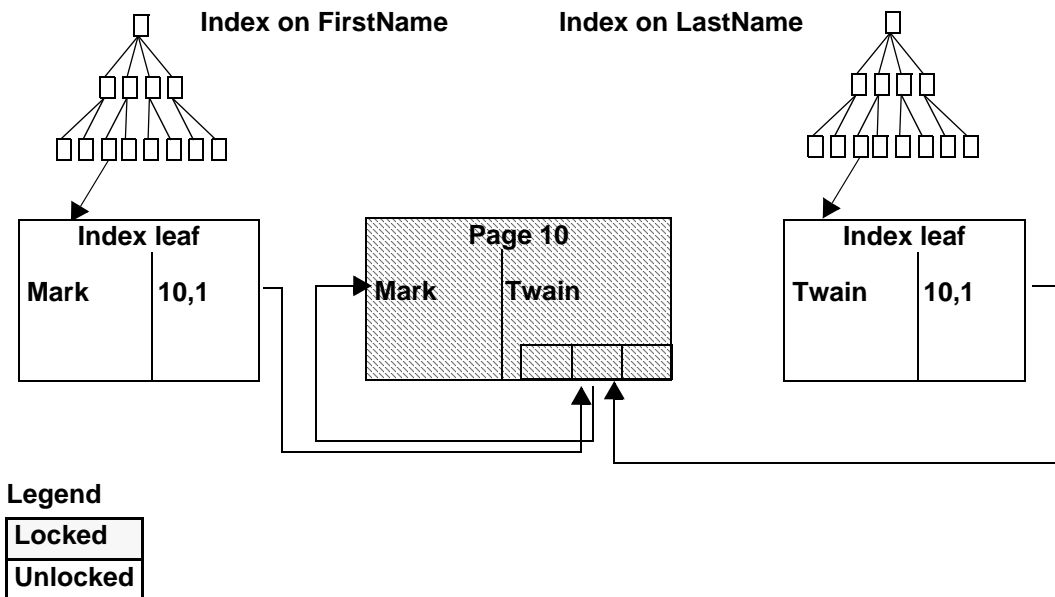
Datapages locking

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. See “Latches” on page 17 and “Choosing a locking scheme based on contention statistics” on page 54 for more information.

Figure 1-2 shows an insert into a datapages-locked table. Only the affected data page is locked.

Figure 1-2: Locks held during datapages locking

insert authors values ("Mark", "Twain")



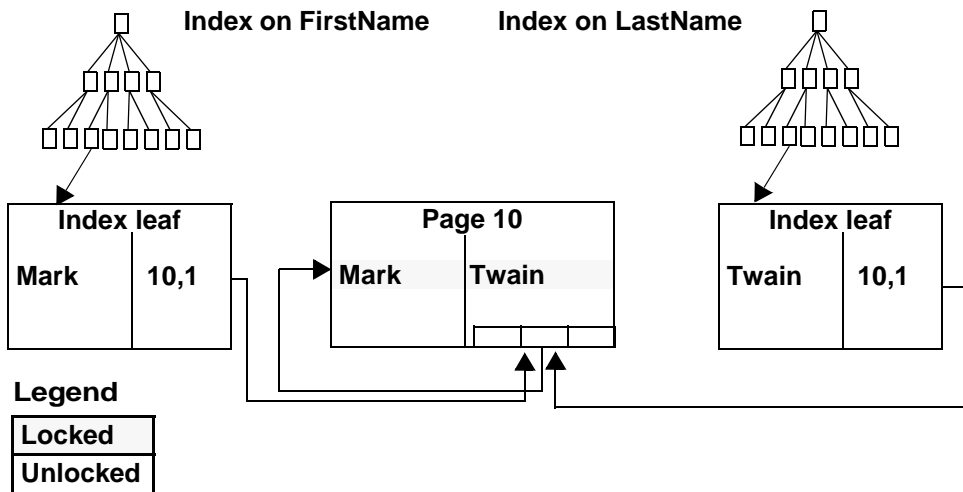
Datarows locking

In datarows locking, row-level locks are acquired on individual rows on data pages. Index rows and pages are not locked. When a row is 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, 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 1-3 shows an insert into a datarows-locked table. Only the affected data row is locked.

Figure 1-3: Locks held during datarows locking

insert authors values ("Mark", "Twain")



Types of locks in Adaptive Server

Adaptive Server has two levels of locking:

- Page locks or table locks are used for tables that use allpages locking or datapages locking.
- Row locks or table locks are used for tables that use datarows locking.

Page or row locks are less restrictive (or smaller) than table locks. A page lock locks all the rows on a data page or an index page; a table lock locks an entire table. 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 is accessed by a statement. Locking strategy is directly tied to the query plan, so a query plan can be as important for its locking strategies as for its I/O implications. For data-only-locked tables, an update or delete statement without a useful index performs a table scan and acquires a table lock. For example, the following statement acquires a table lock if the `account` table uses the `datarows` or `datapages` locking scheme:

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

An update or delete statement using an index begins by acquiring page or row locks. It acquires 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 (configured with `sp_setpglockpromote`). Once table scan 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 44.

Adaptive Server chooses which type of lock to use after it determines the query plan. The way a query or transaction is written can affect the type of lock the server chooses. You can 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 82. Applications can use the `lock table` command to explicitly request a table lock.

Page and row locks

This section describes the types of page and row locks:

- **Shared locks** – Adaptive Server applies **shared locks** 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 a page or row, but no transaction can change data on the page or row while a shared lock exists. Transactions that require an exclusive lock wait for, 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 29.

- **Exclusive locks** – Adaptive Server applies an **exclusive lock** for a data modification operation. When a transaction has 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 1-2 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 1-2: Page locks and row locks

| Statement | Allpages-locked table | Datarows-locked table |
|---|-----------------------|-----------------------|
| <pre>select balance from account where acct_number = 25</pre> | Shared page lock | Shared row lock |

| Statement | Allpages-locked table | Datarows-locked table |
|---|---|---|
| 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 locks followed by exclusive row locks on each affected row |

Table locks

This section describes the types of table locks.

- Intent lock – indicates that page-level or row-level locks are 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 acquiring conflicting table-level locks on the table containing the locked page. An intent lock is held as long as page or row locks are in effect for the transaction.
- Shared lock – similar to a shared page or row 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 – similar to an exclusive page or row lock, except that it affects the entire table. For example, Adaptive Server applies an exclusive table lock during a `create clustered index` command. `update` and `delete` statements on data-only-locked tables require exclusive table locks if their search arguments do not reference indexed columns of the object.

The examples in Table 1-3 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 on `acct_number`.

Table 1-3: Table locks applied during query processing

| Statement | Allpages-locked table | Datarows-locked table |
|--|--|---|
| <code>select balance from account where acct_number = 25</code> | Intent shared table lock Shared page lock | Intent shared table lock Shared row lock |
| <code>insert account values (34, 500)</code> | Intent exclusive table lock Exclusive page lock on data page Exclusive page lock on leaf index pages | Intent exclusive table lock Exclusive row lock |
| <code>delete account where acct_number = 25</code> | 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 |
| <code>update account set balance = 0 where acct_number = 25</code> | Intent exclusive table lock Update page locks followed by exclusive page locks on data pages and leaf-level index pages | With an index on <code>acct_number</code> , intent exclusive table lock Update row locks followed by exclusive row locks on data rows. With no index on a data- only-locked table, 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.

Commands that take intent locks

Versions of Adaptive Server earlier than 15.0.2 used table locks to achieve system catalog synchronization. Adaptive Server version 15.0.2 and later uses intent locks for table-level synchronization and row locks for row-level synchronization. Earlier releases of Adaptive Server locked the entire system catalog while performing operations on the object, so a single lock request was made. However, Adaptive Server version 15.0.2 and later requests locks for all applicable rows while performing operations on the object if there are multiple rows corresponding to an object in a system catalog.

This change means that Adaptive Server version 15.0.2 and later requests more locks to perform the same operation than earlier releases, and increases the number of lock resources the system needs. Consequently, you may need to change the `number of locks` configuration option after you upgrade Adaptive Server.

These commands take intent locks in Adaptive Server version 15.0.2 later when they update a system table:

- create table
- drop table
- create index
- drop index
- create view
- drop view
- create procedure
- drop procedure
- create trigger
- drop trigger
- create default
- drop default
- create rule
- drop rule
- create function
- drop function
- create functional index
- drop functional index
- create computed column
- drop computed column
- select into
- alter table (**all versions**)
- create schema
- reorg rebuild

If two or more of these commands simultaneously access or update the same system table, their intent locks do not conflict with each other so they do not block on the system table.

The `sp_fixindex` and `sp_spaceusage` system procedures provide information about the row-locked catalogs.

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 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 ahead of a queued update task.

After a write transaction has been skipped 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 1-4 on page 14.

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

Adaptive Server uses demand locks to avoid lock starvation for write transactions (when the required number of locks are not available).

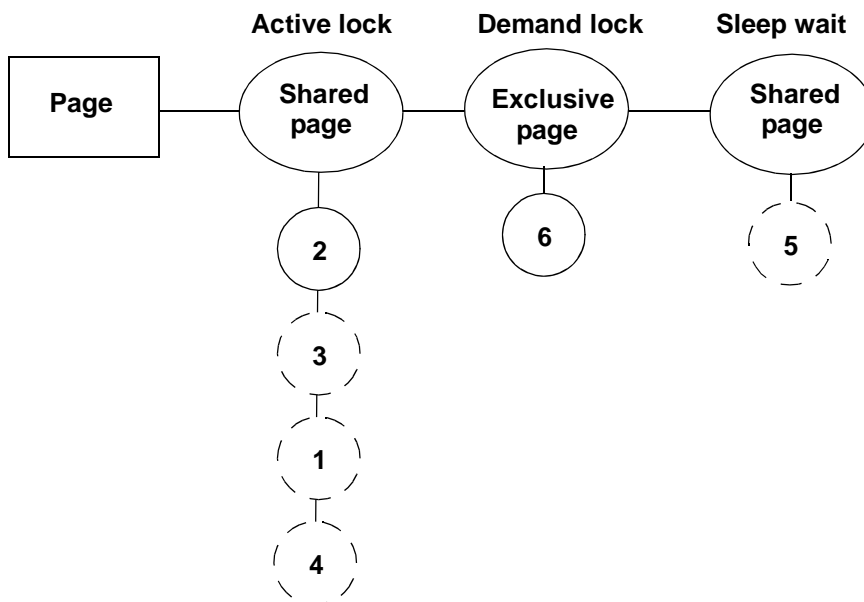
Demand locking with serial execution

Figure 1-4 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 1-4 on page 14:

- 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 1-4: Demand locking with serial query execution



Demand locking with parallel execution

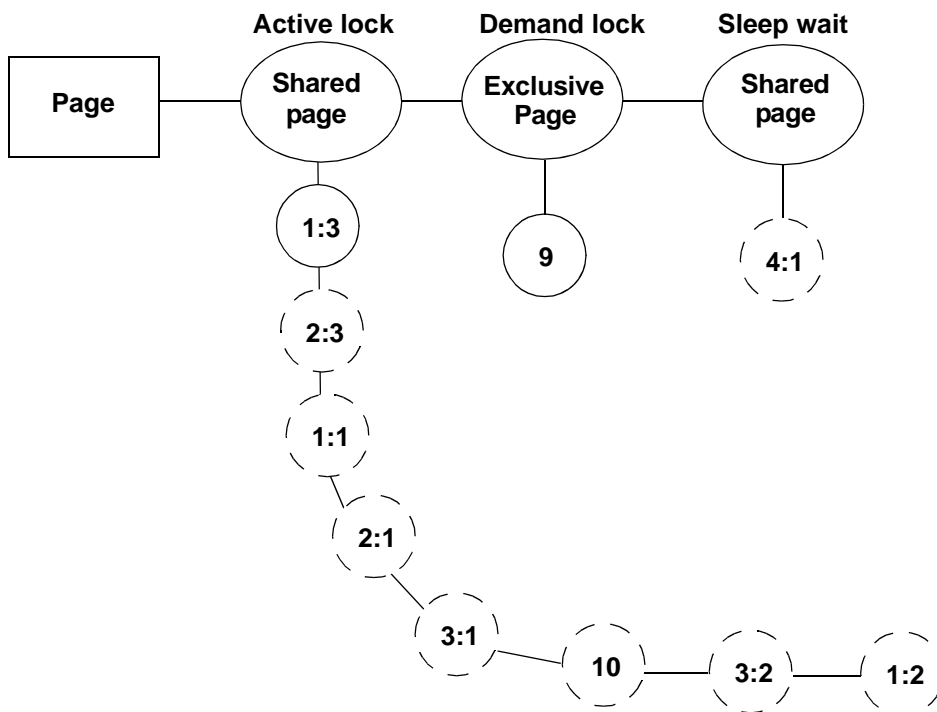
When queries run in parallel, demand locking treats all shared locks from a family of worker processes as though they were a single task. The demand lock permits reads from three families (or a total of three serial tasks and families combined) before granting the exclusive lock.

Figure 1-5 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 1-5:

- 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 1-5: Demand locking with parallel query execution



Row-locked system tables

System tables—except message tables, fake tables (nonrow-oriented tables), and logs—in Adaptive Server version 15.0 and later are row-locked. These tables no longer have a clustered index, but instead have a “placement” index, with a new index ID. Pages at the data level for Adaptive Server are not chained together, and table starting locations are no longer set, but are randomly generated.

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. See “How isolation levels affect locking” on page 19.

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 modify:

- One of the result rows so that it no longer qualifies for the serializable read transaction, by updating or deleting the row
- 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 access the page. Latches are used for datapages and datarows locking, but not for allpages locking.

The most important distinction between a lock and a latch is 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 length of 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 support issues of locking and concurrency:

- Lock compatibility – if a task holds a lock on a page or row, can another task 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 must 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 1-4 summarizes the information about lock compatibility, showing when locks can be acquired immediately.

Table 1-4: Lock compatibility

| If one process has: | Can another process immediately acquire: | | | | |
|--------------------------|--|-----------------|--------------------|-----------------------|---------------------------|
| | A shared lock? | An update lock? | An exclusive lock? | A shared intent lock? | An exclusive intent lock? |
| A shared lock | Yes | Yes | No | Yes | No |
| An update lock | Yes | No | No | N/A | N/A |
| An exclusive lock | No | No | No | No | No |
| A shared intent lock | Yes | N/A | No | Yes | Yes |
| An exclusive intent lock | No | N/A | No | Yes | Yes |

Table 1-5 shows the lock sufficiency matrix.

Table 1-5: 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 summarized in Table 1-6, and are described in more detail on the following pages.

Table 1-6: 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 are 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 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 allpages locking, requesting isolation level 2 also enforces isolation level 3. The Adaptive Server default isolation level is level 1.

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. Table 1-7 shows a select query performing a dirty read.

Table 1-7: Dirty reads in transactions

| 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. | |

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 T3 acquires an exclusive lock on `account`. However, 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 `account` at isolation level 0 before T3 starts, T3 can still acquire its exclusive lock on `account` 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 fewer 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.

During deferred update of a data row, there can be a significant time interval between the `delete` of the index row and the `insert` of the new index row. During this interval, there is no index row corresponding to the data row. If a process scans the index during this interval at isolation level 0, it does not return the old or new value of the data row. See “Deferred updates” in Chapter 1, “Understanding Query Processing” in *Performance and Tuning Series: Query Processing and Abstract Plans*.

`sp_sysmon` reports on these factors. See “Data Cache Management” in *Performance and Tuning Series: Monitoring Adaptive Server with sp_sysmon*.

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 Table 1-8, showing a transaction executed at isolation level 1, to Table 1-7, showing a dirty read transaction.

Table 1-8: Transaction isolation level 1 prevents dirty reads

| 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 |

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. Table 1-9 shows a nonrepeatable read in a transaction at isolation level 1.

Table 1-9: Nonrepeatable reads in transactions

| 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. | |

If transaction T8 modifies and commits the changes to the `account` table after the first query in T7, but before the second one, the same two queries in T7 produce different results. Isolation level 2 blocks 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**. Phantoms 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 Table 1-10, transaction T9, operating at isolation level 1, sees a phantom row in the second query.

Table 1-10: Phantoms in transactions

| 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. | |

If transaction T10 inserts rows into the table that satisfy T9's search condition after 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 concurrency by preventing other transactions from getting their exclusive locks on the data.

Compare the phantom, shown in Table 1-10, with the same transaction executed at isolation level 3, as shown in Table 1-11.

Table 1-11: Avoiding phantoms in transactions

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

In transaction T11, Adaptive Server applies shared page locks and holds the locks until the end of T11. (If `account` is a data-only-locked table, and no index exists on the `acct_number` argument, a shared table lock is acquired.) 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. 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 concurrency by allowing other transactions to obtain 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:

- 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 1-12 shows the types of locks acquired by queries at different isolation levels, for each locking scheme for queries that do not use cursors. Table 1-13 shows information for cursor-based queries.

Table 1-12: 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 | - | - | - | - | No locks are acquired. |
| | | Datapages | - | - | - | - | |
| | | Datarows | - | - | - | - | |
| | 1 | Allpages | IS | S | S | - | * Depends on setting of read committed with lock. See “Locking for select queries at isolation level 1” on page 29. |
| | 2 with noholdlock | Datapages | IS | * | - | - | |
| | 3 with noholdlock | Datarows | IS | - | - | * | |
| | 2 | Allpages | IS | S | S | - | Locks are released at the end of the transaction. See “Isolation level 2 and allpages-locked tables” on page 30. |
| | | Datapages | IS | S | - | - | |
| | | Datarows | IS | - | - | S | |
| select via index scan | 3 | Allpages | IS | S | S | - | Locks are released at the end of the transaction. |
| | 1 with holdlock | Datapages | IS | S | - | - | |
| | 2 with holdlock | Datarows | IS | - | - | S | |
| select via table scan | 3 | Allpages | IS | S | - | - | Locks are released at the end of the transaction. |
| | 1 with holdlock | Datapages | S | - | - | - | |
| | 2 with holdlock | Datarows | S | - | - | - | |
| insert | 0, 1, 2, 3 | Allpages | IX | X | X | - | Locks are released at the end of the transaction. |
| | | Datapages | IX | X | - | - | |
| | | Datarows | IX | - | - | X | |
| writetext | 0, 1, 2, 3 | Allpages | IX | X | - | - | Locks are held on first text page or row; locks released at the end of the transaction. |
| | | Datapages | IX | X | - | - | |
| | | Datarows | IX | - | - | X | |
| delete update any type of scan | 0, 1, 2 | Allpages | IX | U, X | U, X | - | “U” locks are released after the statement completes. “IX” and “X” locks are released at the end of the transaction. |
| | | Datapages | IX | U, X | - | - | |
| | | Datarows | IX | - | - | U, X | |
| delete update via index scan | 3 | Allpages | IX | U, X | U, X | - | “U” locks are released after the statement completes. “IX” and “X” locks are released at the end of the transaction. |
| | | Datapages | IX | U, X | - | - | |
| | | Datarows | IX | - | - | U, X | |
| delete update via table scan | 3 | Allpages | IX | U, X | - | - | Locks are released at the end of the transaction. |
| | | Datapages | X | - | - | - | |
| | | Datarows | X | - | - | - | |

Key: IS intent shared, IX intent exclusive, S shared, U update, X exclusive

Table 1-13: 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) | 0 | Allpages Datapages Datarows | - - - | - - - | - - - | - - - | No locks are acquired. |
| select... for read only | 1 | Allpages | IS | S | S | - | * Depends on setting of read committed with lock. See “Locking for select queries at isolation level 1” on page 29. |
| | 2 with noholdlock | Datapages | IS | * | - | - | |
| | 3 with noholdlock | Datarows | IS | - | - | * | |
| | 2, 3 | Allpages | IS | S | S | - | Locks become transactional after the cursor moves out of the page/row. Locks are released at the end of the transaction. |
| | 1 with holdlock | Datapages | IS | S | - | - | |
| | 2 with holdlock | Datarows | IS | - | - | S | |
| 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 | Allpages | IX | U, X | X | - | Locks become transactional after the cursor moves out of the page/row. Locks are released at the end of the transaction. |
| | 2, holdlock | Datapages | IX | U, X | - | - | |
| | | Datarows | IX | - | - | U, X | |
| select...for update with shared | 2, 3 | Allpages | IX | S, X | X | - | Locks become transactional after the cursor moves out of the page/row. Locks are released at the end of the transaction. |
| | 1 with holdlock | Datapages | IX | S, X | - | - | |
| | 2 with holdlock | Datarows | IX | - | - | S, X | |

Key: IS intent shared, IX intent exclusive, S shared, U update, X exclusive

Lock types during *create index* commands

Table 1-14 describes the types of locks applied by Adaptive Server for create index statements:

Table 1-14: Summary of locks during create index statements

| Statement | Table lock | Data page lock |
|---------------------------|------------|----------------|
| create clustered index | X | - |
| create nonclustered index | S | - |

Key: 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 the inner table are scanned.

select queries on data-only-locked tables first acquire a shared intent table lock. You can configure locking behavior on data pages and data rows issuing 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.

You must declare cursors 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 Chapter 5, “Setting Configuration Parameters” in *System Administration Guide: Volume 1*.

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 is also applied on the allpages-locked tables. Transaction level 2 is 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.

update 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
- Does not contain an `or` clause.

Updates and deletions 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 re-qualified 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 re-qualified 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 insertions when the following conditions are true:

- The table uses datarow locking, and
- The isolation level is 1 or 2.

Under these conditions, scans skip such a row.

The only exception to this rule is if the transaction performing the uncommitted insert was overwriting an uncommitted *delete* of the same row done earlier by the same transaction. In this case, scans block on the uncommitted inserted row.

Skipping uncommitted inserts during *deletes, updates, and inserts*

delete and *update* queries behave the same way as scans do, with regard to uncommitted inserts. When the *delete* or *update* command encounters an uncommitted inserted row with the key value of interest, it skips it without blocking.

The only exception to this rule is if the transaction doing the uncommitted insert was overwriting an uncommitted delete of the same row done earlier by the same transaction. In this case, updates and deletes block on the uncommitted inserted row.

Insert queries, upon encountering an uncommitted inserted row with the same key value, raise a duplicate key error if the index is unique.

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 “Qualifying old and new values for uncommitted updates” on page 36 do not allow the query to proceed, then the query blocks until the lock is released.

For example, transaction T15 in Table 1-15 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. Table 1-15 describes a transaction using pseudo columns, which are columns in the index table that define the parameters of the search and provide access to the results data.

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.

Table 1-15: Pseudo-column-level locking with multiple predicates

| 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 | | |

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 is 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.

Pseudocolumn-level locking

During concurrent transactions that involve `select` 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.
- 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.
- Neither the old nor the 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 Table 1-16 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.

Table 1-16: Pseudo-column-level locking with mutually exclusive columns

| 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. | select lname, fname, phone from accounts where acct_number = 35 commit transaction |
| commit transaction | T14 queries the same row in accounts, but does not access the updated column. T14 does not block. | |

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.

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 the old value, the new value, or both values qualify, the query blocks. In Table 1-17, 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.

Table 1-17: Checking old and new values for an uncommitted update

| 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. | select acct_number, balance from accounts where balance < 50 commit tran |
| commit transaction | T18 queries accounts using an index that includes balance. It does not block since balance does not qualify | |

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 is 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.

Reducing contention

To help reduce lock contention between update and select queries:

- Use datarows or datapages locking for tables with lock contention caused by update and select commands.
- If tables have more than 32 columns, make the first 32 columns the columns 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.

Locking Configuration and Tuning

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

| Topic | Page |
|---|------|
| Locking and performance | 39 |
| Configuring locks and lock promotion thresholds | 44 |
| Choosing the locking scheme for a table | 53 |
| Optimistic index locking | 57 |

Locking and performance

Locking affects the Adaptive Server performance by limiting concurrency. An increase in the number of simultaneous users 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, overall response time and throughput are 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, the throughput of applications is severely affected.

To help reduce deadlock frequency, change the locking scheme to datapages or datarows locking, or redesign the way transactions access data.

- 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, create indexes when there is little activity on your server.

- Turning off delayed deadlock detection causes spinlock contention.

Setting 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 use the deadlock checking period parameter.

Using *sp_sysmon* and *sp_object_stats*

Many of the following sections suggest that you change configuration parameters to reduce lock contention.

Use *sp_object_stats* or *sp_sysmon* to determine whether lock contention is a problem. Then use the stored procedures to determine how tuning to reduce lock contention affects the system.

See “Identifying tables where concurrency is a problem” on page 73 for information on using *sp_object_stats*.

See “Lock management” in *Performance and Tuning Series: Monitoring Adaptive Server with sp_sysmon* for more information about using *sp_sysmon* to view lock contention.

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

Reducing lock contention

Lock contention can impact Adaptive Server throughput and response time. Consider using locking during database design (for example, to avoid joining a high number of tables during queries), and monitor locking during application design.

Address locking contention by 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 deletions and updates.
- Keep transactions short to reduce the time that locks are held.
- Check for contention “hot spots,” especially insertions on allpages-locked heap tables (a heap table is a table that has no clustered index).

Adding indexes to reduce contention

For data-only-locked tables, an `update` or `delete` statement that has no useful index on its search arguments results in a table scan that 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 you cannot create an index for a lengthy `update` or `delete` transaction, you can perform the operation in a cursor, with frequent `commit` transaction statements to reduce the number of page locks.

Keeping transactions short

Keep any transaction that acquires locks as short as possible. In particular, avoid transactions that wait for user interaction while holding locks.

Table 2-1: Examples

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

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

| | |
|----------------------------|---|
| begin tran | <i>isql batch sent to Adaptive Server</i> |
| update account | <i>Locks held waiting for commit</i> |
| set balance = balance + 50 | |
| where acct_number = 25 | |
| go | |
| update account | <i>isql batch sent to Adaptive Server</i> |
| set balance = balance - 50 | <i>Locks released</i> |
| where acct_number = 45 | |
| commit tran | |
| go | |

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 shrink 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 insertions happen on the last page of the page chain.

For example, an unindexed history table that is updated by everyone always has 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 % |

To avoid this:

- 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 using the round-robin strategy. Partitioning a heap table creates multiple page chains in the table, and, therefore, multiple last pages for insertions.

Concurrent inserts to the table are less likely to block one another, since multiple last pages are available. Partitioning improves concurrency for heap tables without creating separate tables for different groups of users.

See “Improving insert performance with partitions” in *Performance and Tuning Series: Physical Database Tuning* for information about partitioning tables.

- Create a clustered index to distribute updates across the data pages in the table.

Like partitioning, this 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 results.

If only a few queries require isolation level 3, use the `holdlock` keyword or the `at isolation serializing` clause in those queries rather than using `set transaction isolation level 3` for the entire transaction.

If most queries in the transaction require isolation level 3, use `set transaction isolation level 3`, but use `noholdlock` or `at isolation read committed` in the queries that can execute at isolation level 1.

- To perform mass insertions, updates, or deletions on active tables, reduce blocking by performing the operation inside a stored procedure using a cursor, with frequent commits.
- If the application must return a row, wait for user interaction, and then update the row, consider using timestamps and the `tsequal` function rather than `holdlock`.

- If you use third-party software, check the locking model in applications carefully for concurrency problems.

Other tuning efforts can also 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, the process holds the lock much longer than it would if the additional page were already in cache. In this case, better cache utilization or the use of large I/O can reduce lock contention. You can also reduce lock contention by improving indexing and distributing physical I/O evenly 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 hash table and the number of spinlocks that protect the page/row lock hash table, table lock hash table, and address lock hash table
- 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

Configuring the Adaptive Server lock limit

By default, Adaptive Server is configured with 5000 locks. System Administrators can use `sp_configure` to change this limit. For example:

```
sp_configure "number of locks", 25000
```

You may also need to adjust the `sp_configure` parameter `max 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.

Start with 20 locks for each active concurrent connection, plus 20 locks for each worker process. Consider increasing the number of locks if:

- You use datarows locking
- Queries run at isolation level 2 or 3, or use `serializable` or `holdlock`
- Parallel query processing is enabled, especially for isolation level 2 or 3 queries
- You perform many multirow updates
- 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” on page 89 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 requires 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 (lock manager)

Table 2-2: lock hashtable size

| Summary information | Value |
|---------------------|----------------------|
| Default | 2048 |
| Range | 1 – 2147483647 |
| Status | Static |
| Display level | Comprehensive |
| Required role | System Administrator |

The `lock hashtable size` parameter specifies the number of hash buckets in the lock hash table. This table manages all row, page, and table locks, and all lock requests. Each time a task acquires a lock, the lock is assigned to a hash bucket, and each lock request for that lock checks the same hash bucket. Setting this value too low results in large numbers of locks in each hash bucket and slows the searches.

On Adaptive Servers with multiple engines, setting this value too low can also lead to increased spinlock contention. Do not set the value to less than the default value, 2048. `lock hashtable size` must be a power of 2. If the value you specify is not a power of 2, `sp_configure` rounds the value to the next highest power of 2 and prints an informational message.

The optimal hash table size is a function of the number of distinct objects (pages, tables, and rows) that will be locked concurrently. The optimal hash table size is at least 20 percent of the number of distinct objects that need to be locked concurrently. See “Lock management” in *Performance and Tuning Series: Monitoring Adaptive Server with sp_sysmon* for more information on configuring the lock hash table size.

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, use `sp_sysmon` to check the average hash chain length at peak periods. If the average length of the hash chains exceeds 4 or 5, consider increasing the value of `lock hashtable size` from its current setting to the next power of 2.

The hash chain length may be high during large insert batches, such as bulk copy operations. This is expected behavior, and does not require you to reset `lock hashtable 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 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 scan session may scan data from more than one partition. Lock promotion is based on the number of page or row locks acquired across all the partitions accessed in the scan.

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

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.

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 can acquire a table lock. The default value is 200. Adaptive Server cannot 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. The possibility of all locks being exhausted is especially high with queries that update a large number of rows in a datarows-locked table, or that 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.

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.

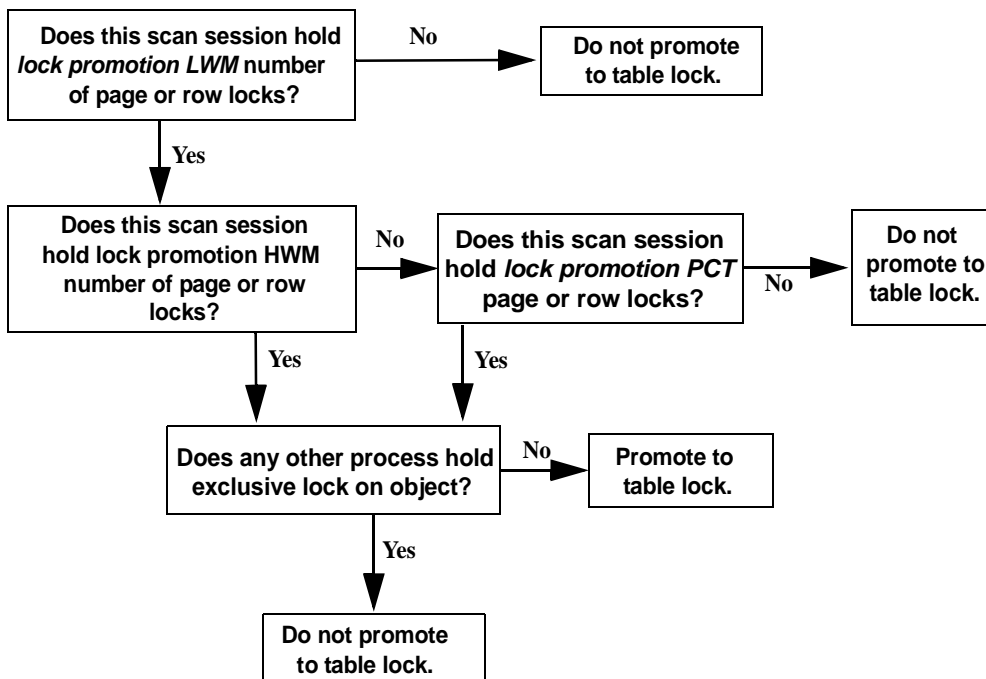
The default value is 100.

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:

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

Setting lock promotion PCT to a very low value increases the chance of a particular user transaction acquiring a table lock. Figure 2-1 shows how Adaptive Server determines whether to promote page locks on a table to a table lock.

Figure 2-1: 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:

```
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 for 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.

Dropping database and table settings

To remove table or database lock promotion thresholds, use `sp_droplockpromote` 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 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.

If there is a problem, look for signs of lock contention in the “Granted” and “Waited” data in the “Lock Detail” section of `sp_sysmon` output.

See “Lock promotions” and “Lock detail” in *Performance and Tuning Series: Monitoring Adaptive Server with sp_sysmon* for more information.

If lock contention is high and lock promotion is frequent, consider changing the lock promotion thresholds for the tables involved.

Use Monitor Server or the monitoring tables 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, choose a lock scheme for a new table based on the likelihood that applications will experience lock contention on the table. Whether to change the locking scheme for an existing table can be based on contention measurements on the table, but should also take application performance into account.

Here are some typical situations and general guidelines for choosing the locking scheme:

- Applications require clustered access to data rows due to range queries or order by clauses. Allpages locking provides more efficient clustered access than data-only-locking. Rows may not be returned in key order of the clustered index for queries.

against a data-only-locked table without the use of an order by clause..

- A large number of applications access 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 tables that have 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 existing applications experience blocking and deadlock problems, 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 tables involved in the deadlock, either using `sp_object_stats` or by enabling the print deadlock information configuration parameter.
- 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 56.
- 3 If the table uses allpages locking, convert the locking scheme to datapages locking to determine whether that solves the concurrency problem.
- 4 Re-run the 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. 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-contenting 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 `number of locks` configuration parameter.

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 45.

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.

Re-running `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 only those tables with the highest lock contention. Then test the results of these changes by re-running `sp_object_stats`.

Run your usual performance monitoring tests both before and after you make the changes.

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:
 - 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 insertion and forwarded rows.

Performance remains high for tables that do not experience many insertions. On tables that get many insertions, 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 columns, subtract 2 bytes for each variable-length column (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.

Optimistic index locking

Optimistic index locking can resolve increased contention on some important resources, such as the spinlocks that guard address locks on the root page of an index partition.

Applications where this amount of contention might occur are typically those in which:

- Access to a specified index constitutes a significant portion of the transaction profile, and many users are concurrently executing the same workload.
- Different transactions, such as ad hoc and standardized queries, use the same index concurrently.

Optimistic index locking does not acquire an address lock on the root page of an index partition during normal data manipulation language (DML) operations. If your updates and insertions can cause modifications to the root page of the accessed index partition, optimistic index locking restarts the search and acquires an exclusive table lock, not an address lock.

Two stored procedures are changed by optimistic index locking:

- `sp_chgattribute` enables or disables optimistic index locking; when enabled, setting an exclusive table lock on the table you specify.
- `sp_help` includes a column that displays optimistic index lock.

For more information, see the *Adaptive Server Reference Manual: Procedures*.

Using optimistic index locking

Use optimistic index locking when any or all of the following conditions are true:

- There is significant contention on the lock address hash bucket spinlock.
- None of the indexes on this table cause modifications to the root page.
- The number of index levels is high enough to cause no splitting or shrinking of the root page.
- There are large numbers of concurrent accesses to read-only tables on heavily trafficked index pages.
- A database is read-only.

Cautions and issues

Since an exclusive table lock blocks all access by other tasks to the entire table, you must thoroughly understand the user access patterns of your application before enabling optimistic index locking.

The following circumstances require an exclusive table lock:

- Adding a new level to the root page
- Shrinking the root page
- Splitting or shrinking the immediate child of the root page, causing an update on the root page

Do not use optimistic index locking when:

- You have small tables (that are not read-only) with index levels no higher than 3.

- You envision possible modifications to the root page of an index

Note An exclusive table lock is an expensive operation, since it blocks access to the entire table. Use extreme caution in setting the optimistic index locking property.

Locking Reports

This chapter discusses tools that report on locks and locking behavior.

| Topic | Page |
|---|------|
| Locking tools | 61 |
| Deadlocks and concurrency | 67 |
| Identifying tables where concurrency is a problem | 73 |
| Lock management reporting | 74 |

Locking tools

`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

`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.

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

For example, consider what happens if you run three sessions in the `pubs2` database: session one deletes the `authors` table, session two selects all the data from the `authors` table, and the third session running `sp_who` against `spid 15`. In this situation, session two hangs, and session three reports this in the `sp_who` output:

```
1> sp_who "15"
```

```

fid      spid      status      loginame      origname      hostname      blk_spid
dbname      cmd      block_xloid
-----
0        15        recv sleep  sa          sa          PSALDINGXP    0
pubs2    AWAITING COMMAND  0

```

If you run `sp_who` against spid 16:

```

sp_who "16"
fid      spid      status      loginame      origname      hostname      blk_spid
dbname      cmd      block_xloid
-----
0        16        lock sleep  sa          sa          PSALDINGXP    15
pubs2    SELECT

```

If you run `sp_lock` against spid 15, the `class` column displays the cursor name for locks associated with the current user's cursor and the cursor ID for other users:

```

1> sp_lock 15
fid      spid      loid      locktype      table_id      page
row      dbname      class      context
-----
0        15        30        Ex_intent      576002052     0
0        pubs2      Non Cursor Lock
0        15        30        Ex_page-blk    576002052     1008
0        pubs2      Non Cursor Lock
0        15        30        Ex_page        576002052     1040
0        pubs2      Non Cursor Lock      Ind pg

```

If you run `sp_lock` against spid 16, the `class` column displays the cursor name for locks associated with the current user's cursor and the cursor ID for other users:

```

fid      spid      loid      locktype      table_id
page      row      dbname      class context
-----
0        16        32        Sh_intent      576002052

```

0 0 pubs2 Non Cursor Lock

Note The sample output for `sp_lock` and `sp_familylock` 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 |

This example shows the lock status of serial processes and one parallel process:

- `spid 15` holds an exclusive intent lock on a table, one data page lock, and two index page locks. 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.
- `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 `spids` 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.

The `lock_type` 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 “demand” suffix indicates that the process will acquire an exclusive lock as soon as all current shared locks are released.

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 family duration 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 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 multiple `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 example shows that the coordinating process (fid 51, spid 51) holds a shared intent lock on each of four tables and a 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, row, or table and each wants to acquire a lock on the page, row, or table locked by the other process. When this happens, the first process is waiting for the second to release the lock, but the second process will not release it until the lock held by the first process is released.

Server-side versus application-side deadlocks

When tasks deadlock in Adaptive Server, a deadlock detection mechanism rolls back one of the transactions, and sends messages to the user and to the Adaptive Server error log. Sometimes application-side deadlock situations arise 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 deletions 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

Below is an example of a deadlock between two processes.

| T19 | Event sequence | T20 |
|--|---|---|
| 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 T20 to release its lock while T20 waits for T19 to release its lock; deadlock occurs. | update savings set balance = balance + 75 where acct_number = 25 |
| commit transaction | | commit transaction |

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. Message 1205 indicates that a user transaction has been 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 they can still 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.
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 that occur 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.

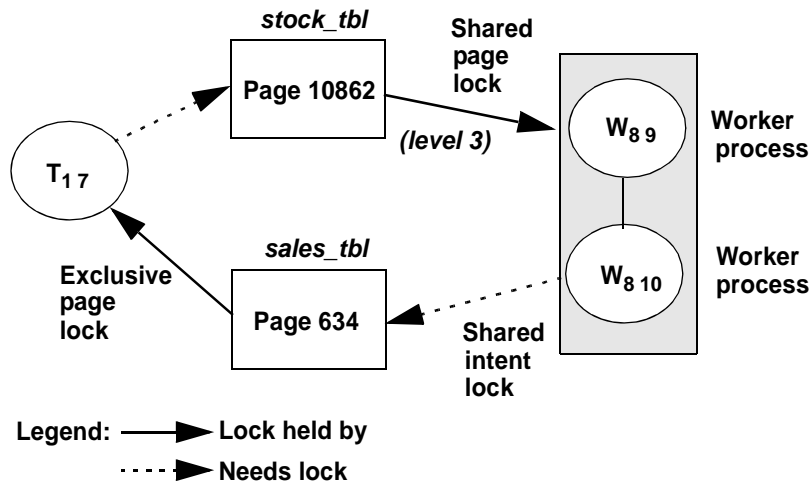
If another worker process in the same family needs a lock on `tableA`, the result is a deadlock. Figure 3-1 illustrates the following deadlock scenario:

- 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 (T17) is performing inserts to `stock_tbl` and `sales_tbl` in a transaction.

These are the steps that lead to the deadlock:

- W8 9, a worker process with a `fid` of 8 and a `spid` of 9, holds a shared lock on page 10862 of `stock_tbl`.
- T17 holds an exclusive lock on page 634 of `sales_tbl`. T17 needs an exclusive lock on page 10862, which it cannot acquire until W8 9 releases its shared lock.
- The worker process W8 10 needs a shared lock on page 634, which it cannot acquire until T17 releases its exclusive lock.

Figure 3-1: A deadlock involving a family of worker processes



Printing deadlock information to the error log

Adaptive Server detects server-side deadlocks to the application and reports them in the server's error log. The error message sent to the application is error 1205.

To get information about the tasks that deadlock, set the `print deadlock information` configuration parameter to 1. This setting sends more detailed deadlock messages to the log and to the terminal session where the server started.

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.)

However, setting `print deadlock information` to 1 can degrade Adaptive Server performance. For this reason, use it only to determine the cause of deadlocks.

Disabling print deadlock information (setting it to 0) means that Adaptive Server does not send any information about deadlocks to the error log.

The deadlock messages contain detailed information, including:

- 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. SQL Text select * from authors where au_id like '172%'
```

```
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

Deadlocks may occur when many long-running transactions are executed at the same time in the same database. Deadlocks become more common as lock contention increases between 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 2, “Locking Configuration and Tuning.”

Acquiring 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 3-1 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). 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. Use the `deadlock checking period` configuration parameter to specify the minimum amount of time (in milliseconds) that a process waits before it initiates a deadlock check.

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

Identifying tables where concurrency is a problem

`sp_object_stats` prints table-level information about lock contention. Use it to:

- Report on 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. The default is 10. 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 3-1 shows the meaning of the values.

Table 3-1: 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 |
| Wait-time | The total number of milliseconds that all tasks spent waiting for a lock |
| Contention | The percentage of times that a task had to wait or encountered a deadlock |

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

Output from sp_sysmon provides 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” in *Performance and Tuning Series: Monitoring Adaptive Serve with sp_sysmon*.

Use Monitor Server and the monitoring tables to pinpoint locking problems. For more information, see the *Performance and Tuning Series: Monitoring Tables* and the *Monitor Server User Guide*.

Using Locking Commands

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

| Topic | Topic |
|---|-------|
| Specifying the locking scheme for a table | 77 |
| Controlling isolation levels | 82 |
| Readpast locking | 86 |
| Cursors and locking | 87 |
| Additional locking commands | 89 |

Specifying the locking scheme for a table

The locking schemes in Adaptive Server provide the flexibility to choose the best locking scheme for each table in an 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` – specifies a server-wide default locking scheme
- `create table` – specifies the locking scheme for newly created tables
- `alter table` – changes the locking scheme for a table to any other locking scheme
- `select into` – specifies the locking scheme for a table created by selecting results from other tables

Specifying a server-wide locking scheme

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 data pages:

```
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*

Use `create table` to specify the locking scheme for a new table. 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 the 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.

Changing a locking scheme with *alter table*

Use `alter table` 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 you to copy the data rows to new pages and re-create any indexes on the table.

Changing the locking scheme 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.

If you are changing from `datapages` locking to `datarows` locking or vice versa you need not copy data pages and rebuild indexes. Switching between data-only locking schemes updates only system tables, and finishes quickly.

Note You cannot use data-only locking on 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 “Determining Sizes of Tables and Indexes” in *Performance and Tuning Series: Physical Database Tuning* 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, Sybase recommends that you take these steps:

- If the table is partitioned, and you have not run `update statistics` since making 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.

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 “Setting Space Management Properties” in *Performance and Tuning Series: Physical Database Tuning* for information on rows and row overhead.
- Determine if there is enough space. See “Determining the space available for maintenance activities” in *Performance and Tuning Series: Physical Database Tuning* .
- 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.
 - Configure for optimum parallel sort performance.

After alter table completes:

- Run `dbcc checktable` on the table and `dbcc checkalloc` on the database to ensure database consistency.
- Perform a database dump.

Note After you have changed the locking scheme from allpages locking to data-only locking or vice versa, you cannot use `dump transaction` to back up the transaction log.

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. Most of the cost comes from the I/O required to copy the tables and re-create the indexes. Some logging is also required.

When moving from allpages to data-only locking or from data-only to allpages locking, `alter table ... lock:`

- 1 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 fewer than 10 bytes are padded to 10 bytes during this step. If you are changing to allpages locking from data-only locking, padding is stripped from rows of fewer than 10 bytes.

- 2 Drops and re-creates all indexes on the table.
- 3 Deletes the old set of table pages.
- 4 Updates the system tables to indicate the new locking scheme.
- 5 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.

Sort performance during *alter table*

During `alter table`, 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:

- Dropping the nonclustered indexes
- Altering the locking scheme
- Configuring for best parallel sort performance
- Re-creating two or more nonclustered indexes at once

Specifying a locking scheme with *select into*

You can specify a locking scheme when you create a new table using `select into`. 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 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 required results with the least blocking.

Note If you use transaction isolation level 2 (repeatable reads) on allpages-locked tables, isolation level 3 (serializing reads) is also enforced.

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 using:

```
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.

You can display the current isolation level for a session with the global variable `@@isolation`.

Syntax for query-level and table-level locking options

You can specify the `holdlock`, `noholdlock`, and `shared` options for each table in a `select` or `readtext` 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 <code>select</code> statements in cursors open for update. |

These keywords affect locking for the transaction: if you use `holdlock`, 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 | <code>read uncommitted</code> | Reads uncommitted changes; use from level 1, 2, or 3 queries to perform dirty reads (level 0). |
| 1 | <code>read committed</code> | Reads only committed changes; wait for locks to be released; use from level 0 to read only committed changes, but without holding locks. |
| 2 | <code>repeatable read</code> | Holds shared locks until the transaction completes; use from level 0 or level 1 queries to enforce level 2. |
| 3 | <code>serializable</code> | 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
```

Making locks more restrictive

If isolation level 1 is sufficient for most 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, row, or table lock more restrictive. `holdlock` 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, row, 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.

If the session isolation level is 0, and only committed changes must be read from 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 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 improve concurrency by specifying `noholdlock` with those queries.

For example, if the transaction isolation level is set to 3, which normally causes 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
```

If the 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.

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 88 for more information.

Readpast locking

Readpast locking allows `select` and `readtext` queries to skip all rows or pages locked with incompatible locks. The queries do not block, terminate, or return error or advisory messages to the user. Readpast locking 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 “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. For data-only-locked tables, Adaptive Server may use a table scan to avoid the Halloween problem. For more information see “Optimization for Cursors” in *Performance and Tuning Series: Query Processing and Abstract Plans*.

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.

Locking behavior for cursors at each isolation level is as follows:

- 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 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 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

In transactions, you can use the `lock table` command to:

- 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.

Lock timeouts

You can specify the amount of 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 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.

Use `sp_sysmon` to monitor the number of tasks that exceed the time limit while waiting for a lock.

See “Lock time-out information” in *Performance and Tuning Series: Monitoring Adaptive Server with sp_sysmon*.

Indexes

This chapter describes how Adaptive Server stores indexes and uses them to speed data retrieval for select, update, delete, and insert operations.

| Topic | Page |
|---|------|
| Types of indexes | 92 |
| Indexes and partitions | 95 |
| Clustered indexes on allpages-locked tables | 96 |
| Nonclustered indexes | 105 |
| Index covering | 111 |
| Indexes and caching | 114 |

Indexes are the most important physical design element in improving database performance:

- Indexes help to avoid table scans. A few index pages and data pages can satisfy many queries without requiring reads on hundreds of data pages.
- For some queries, data can be retrieved from a nonclustered index without accessing data rows.
- Clustered indexes can randomize data inserts, avoiding insert hot spots on the last page of a table.
- Indexes can help to avoid sorts, if the index order matches the order of the columns in an `order by` clause.
- For most partitioned tables, you can create global indexes with one index tree to cover the whole table, or you can create local indexes with multiple index trees, each of which covers one partition of the table.

In addition to their performance benefits, indexes can enforce the uniqueness of data.

Indexes are database objects created on a table to speed direct access to specific data rows. Indexes store the values of the keys named when the index was created and logical pointers to the data pages or to other index pages.

Although indexes speed data retrieval, they can slow down data modifications, since most changes to the data require index updates. Optimal indexing demands an understanding of:

- The behavior of queries that access unindexed heap tables, tables with clustered indexes, and tables with nonclustered indexes
- The mix of queries that run on your server
- The relative benefits of local and global indexes on partitioned tables
- The Adaptive Server optimizer

Types of indexes

Adaptive Server provides two general types of indexes that can be created at the table or at the partition level.

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

You can create only one clustered index on a table or partition 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 2, “Data Storage,” in *Performance and Tuning Series: Physical Database Tuning* discusses heaps and SQL operations on heaps.

For partitioned tables, indexes may be either **local** or **global** (see “Indexes and partitions” on page 95).

Function-based indexes are a type of nonclustered index which use one or more expressions as the index key. See the *Transact-SQL User’s Guide* for more on creating function-based indexes. See also Chapter 6, “Indexing for Concurrency Control,” for information on when to use function-based indexes.

Index pages

Index entries are stored as rows on index pages in a format similar to that of data rows on data pages. Index entries store 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 typically much more densely populated than data pages. If a data row has 200 bytes (including row overhead), there are 10 rows per page on a 2K server. However, an index on a 15-byte field has about 100 rows per index page on a 2K server (the pointers require 4 – 9 bytes per row, depending on the type of index and the index level).

Indexes can have multiple levels:

- Root level
- Leaf level
- Intermediate level

Root level

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

Data-only-locked tables always have a leaf level between the root page and the data pages.

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

Leaf level

The lowest level of the index is the leaf level. At the leaf level, an 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, and the row containing the specific key value.

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. Indexes on a very small table may not have an intermediate level; the root pages point directly to the leaf level.

Index size

Table 5-1 describes the limits for index size for APL and DOL tables:

Table 5-1: Index row-size limit

| Page size | User-visible index row-size limit | Internal index row-size limit |
|-------------------|-----------------------------------|-------------------------------|
| 2K (2048 bytes) | 600 | 650 |
| 4K (4096 bytes) | 1250 | 1310 |
| 8K (8192 bytes) | 2600 | 2670 |
| 16K (16384 bytes) | 5300 | 5400 |

You can create tables with columns wider than the limit for the index key; however, these columns become nonindexable. For example, if you perform the following on a 2K page server, then try to create an index on c3, the command fails and Adaptive Server issues an error message because column c3 is larger than the index row-size limit (600 bytes).

```
create table t1 (  
  c1 int  
  c2 int  
  c3 char(700))
```

You can still create statistics for a nonindexable column, or include it in search results. Also, if you include the column in a *where* clause, it is evaluated during optimization.

An index row size that is too large can result in frequent index page splits. Page splits can make the index level grow linearly with the number of rows in the table, making the index useless because the index traverse becomes expensive. Adaptive Server limits the index size to, at most, approximately one third of server's page size, so that each index page contains at least three index rows.

Indexes and partitions

Partitioned tables include additional indexing options. Indexes on partitioned tables can be either global (one index tree covering all the data in the table) or local (multiple index trees, each of which covers only the data within its corresponding data partition).

Local indexes on partitioned tables

Both clustered and nonclustered local indexes are supported on all types of partitioned tables. Each index partition spans a single data partition; that is, the index partition is “equipartitioned” with the table. On range-, list-, and hash-partitioned tables, clustered indexes are always local indexes. When you create a local index, you actually create separate index trees for each partition in the table. However, Adaptive Server does not support partial indexes, so you cannot selectively create local indexes for certain partitions.

Global indexes on partitioned tables

Global indexes on partitioned tables span all the partitions in the table; that is, a single index tree covers all the data in the table, regardless of partitions. Global indexes on range-, list-, or hash-partitioned tables may only be nonclustered, since clustered index ordering conflicts with partition ordering of the data.

Global clustered indexes are allowed on round-robin partitioned tables.

Local versus global indexes

- Local indexes can increase concurrency through multiple index access points, which reduces root-page contention.
- You can place local nonclustered index subtrees (index partitions) on separate segments to increase I/O parallelism.
- You can run `reorg rebuild` on a per-partition basis, reorganizing the local index sub-tree while minimizing the impact on other operations.
- Global nonclustered indexes are better for covered scans than local indexes, especially for queries that need to fetch rows across partitions.

Unsupported partition index types

- Global partitioned indexes are not supported, meaning that global indexes that cover all the data in the table are not themselves partitioned.
- Global clustered indexes are supported only on round-robin partitioned tables.

Clustered indexes on allpages-locked tables

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

Physical ordering means that:

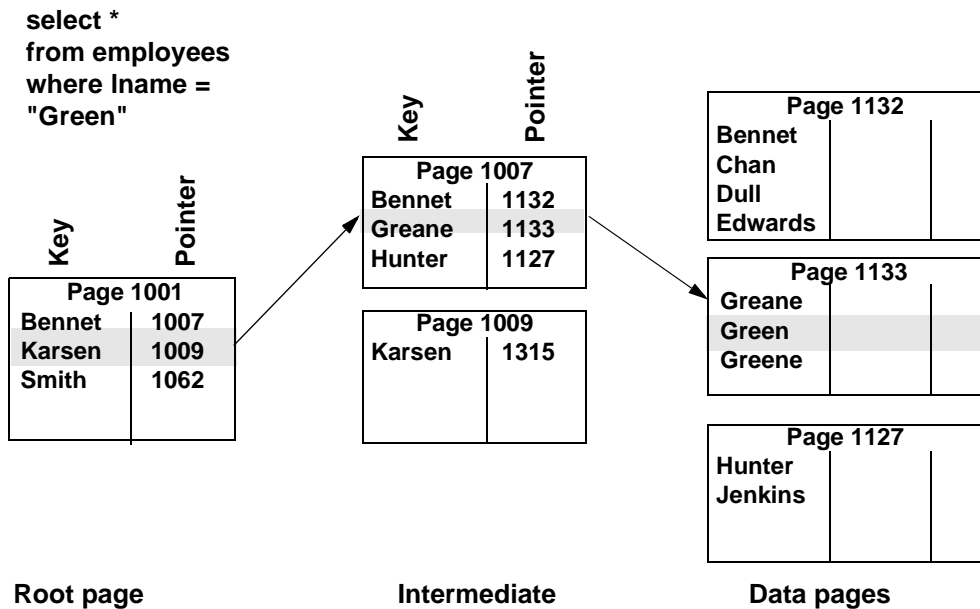
- All entries on a data page are in index key order.
- By following the “next page” pointers on the data pages, Adaptive Server reads the entire table in index key order.

On the root and intermediate pages, each entry points to a page on the next level.

Clustered indexes and select operations

Adaptive Server uses `syspartitions` to find the root page to select a particular column (for example, a last name) using a clustered index (in versions earlier than 15.0, Adaptive Server used `sysindexes`). Adaptive Server 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.

Figure 5-1: Selecting a row using a clustered index, allpages-locked table



In Figure 5-1, 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 using the clustered index requires one read for each of the:

- Root level of the index
- Intermediate level
- Data page

These reads may come either from cache or from disk. 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.

Relationship between physical and logical reads

If Adaptive Server reads a page from disk, it is counted as a physical and a logical read. The cost of 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.

Clustered indexes and insert operations

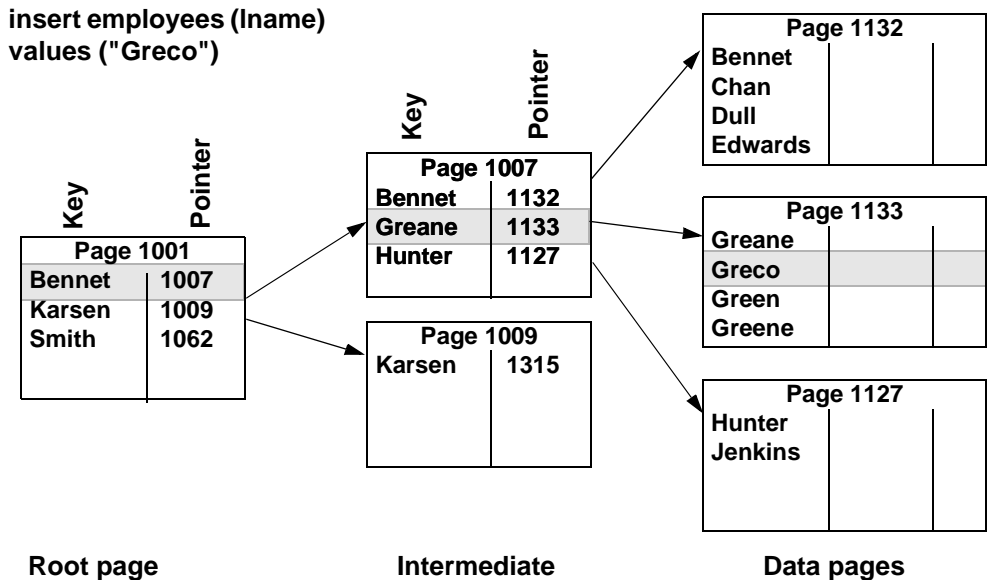
When you insert a row into an allpages-locked table with a clustered index, the data row must be placed in physical order according to the key value on the table.

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 insertion does not affect any other pages in the database.

The clustered index is used to find the location for the new row.

Figure 5-2 shows a simple case where there is room on an existing data page for the new row. In this case, the key values in the index do not need to change.

Figure 5-2: Inserting a row into an allpages-locked table with a clustered index



Page splitting on full data pages

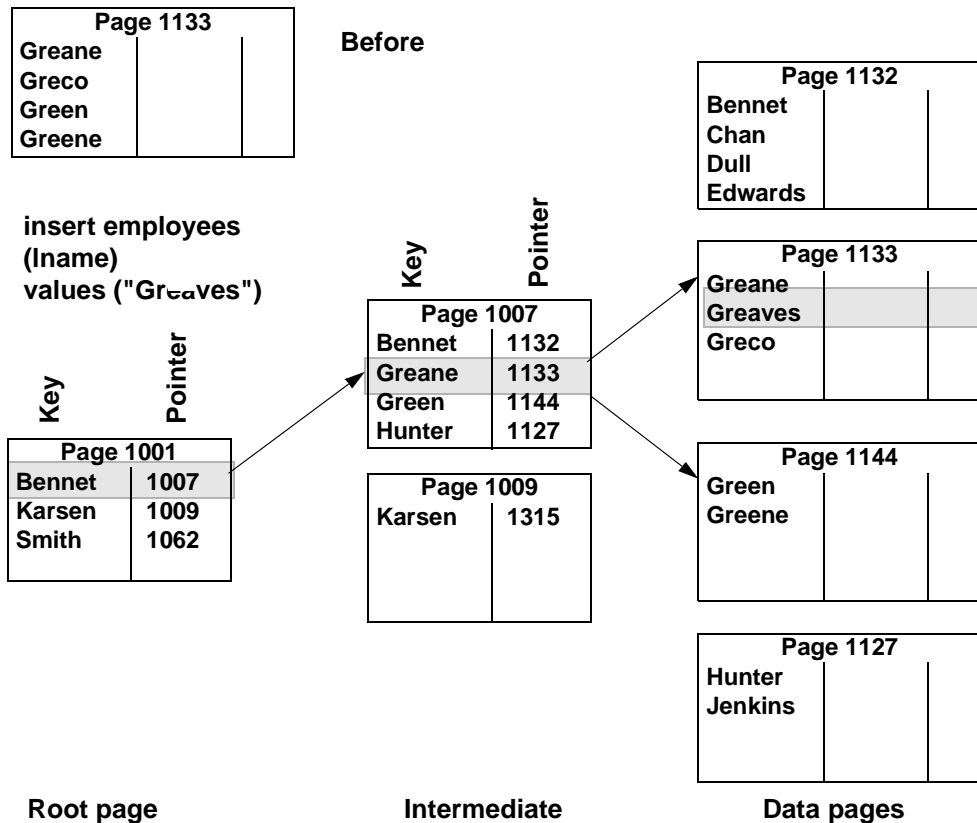
If there is not enough room on the data page for the new row, a page split must be performed.

- 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 100.

In Figure 5-3, the page split requires adding a new row to an existing index page, page 1007.

Figure 5-3: Page splitting in an allpages-locked table with a clustered index



Exceptions to page splitting

There are exceptions to 50-50 page splits:

- If you insert a large row that cannot fit on the page before or the page after the page that requires splitting, two new pages are allocated, one for the large 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
- Page pointers on adjacent pages to maintain page linkage
- 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.

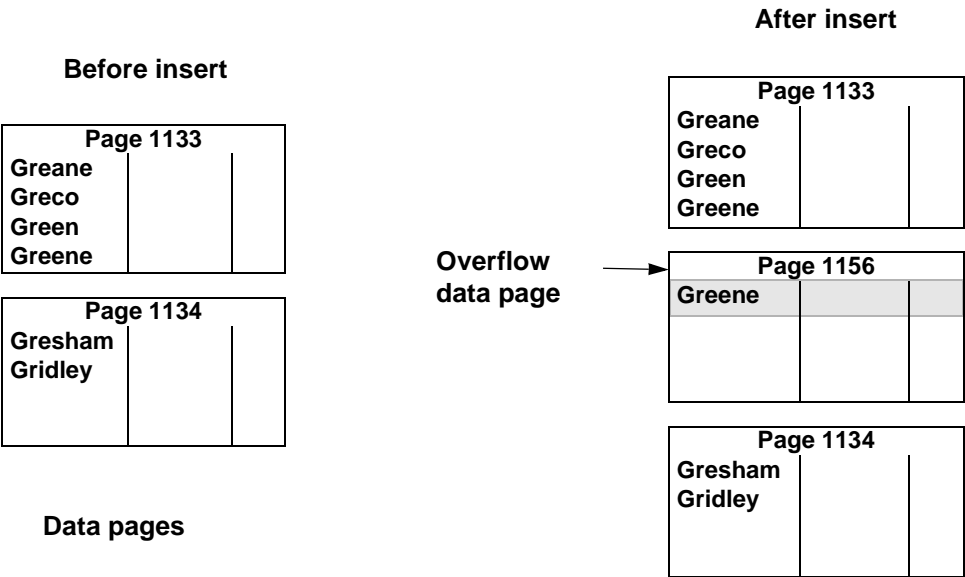
See “Choosing space management properties for indexes” on page 140.

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.

Figure 5-4: Adding an overflow page to a clustered index, allpages-locked table

**insert employees (lname)
values("Greene")**



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

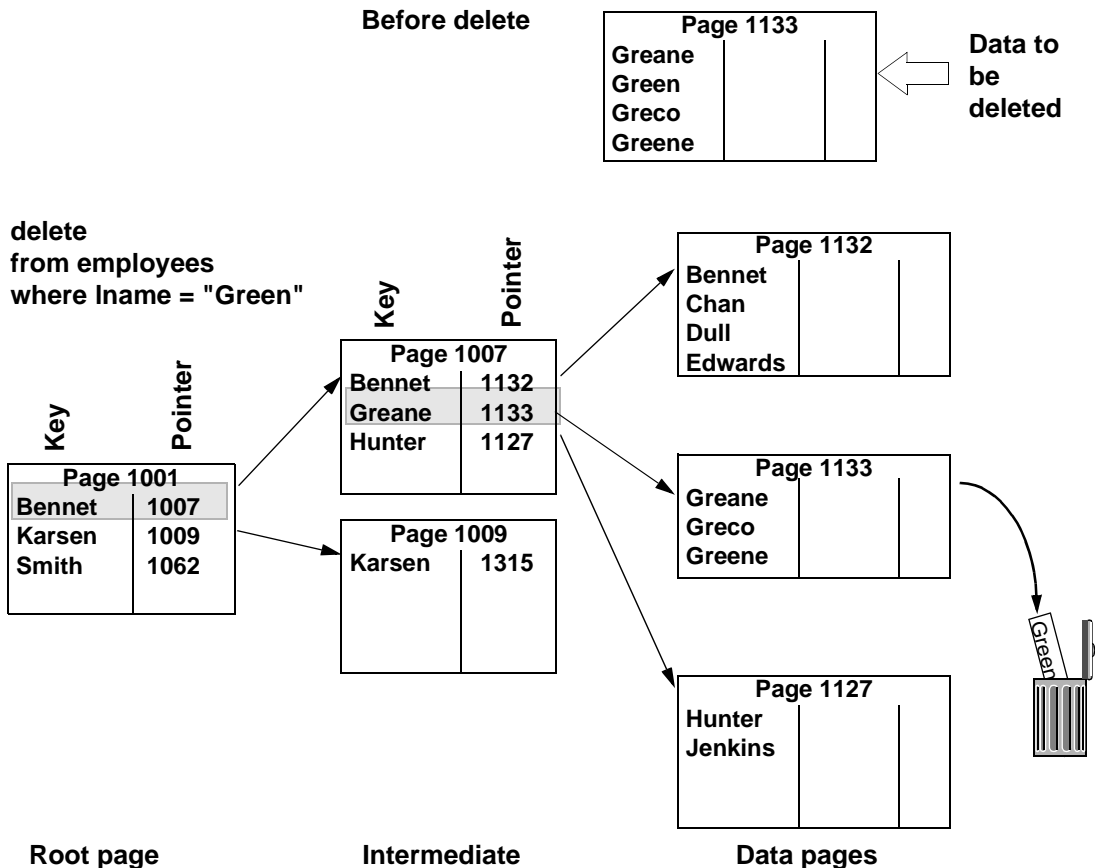
The clustered index does not contain pointers directly to overflow pages. Instead, the next page pointers are used to follow the chain of overflow pages until a value is found that does not match the search value.

Clustered indexes and delete operations

When you delete a row from an allpages-locked table that has a clustered index, other rows on the page move up to fill the empty space so that the data remains contiguous on the page.

Figure 5-5 shows a page that has four rows before a delete operation removes the second row on the page. The two rows that follow the deleted row are moved up.

Figure 5-5: Deleting a row from a table with a clustered index



Deleting the last row on a page

If you delete the last row on a data page, the page is deallocated and the next and previous page pointers on the adjacent pages are changed.

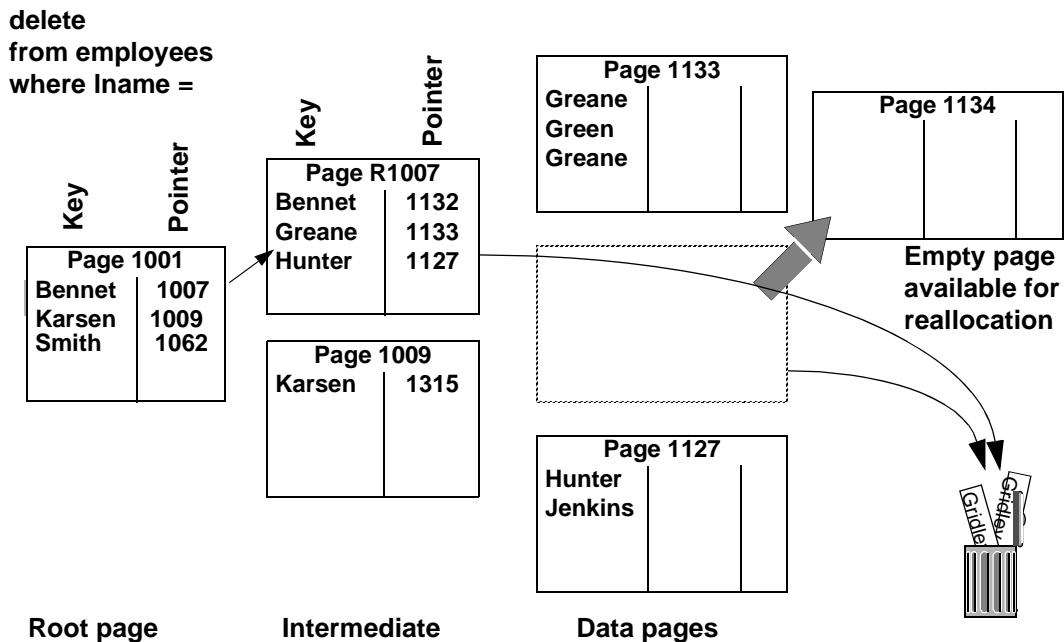
The rows that point to that page in the leaf and intermediate levels of the index are removed.

If the deallocated data page is on the same extent as other pages belonging to the table, it can be used again when that table needs an additional page.

If the deallocated data page is the last page on the extent that belongs to the table, the extent is also deallocated and becomes available for the expansion of other objects in the database.

In Figure 5-6, which shows the table after the deletion, the pointer to the deleted page has been removed from index page 1007 and the following index rows on the page have been moved up to keep the used space contiguous.

Figure 5-6: Deleting the last row on a page (after the delete)



Index page merges

If you delete a pointer from an index page, leaving only one row on that page, the row is moved onto an adjacent page, and the empty page is deallocated. The pointers on the parent page are updated to reflect the changes.

Nonclustered indexes

The B-tree works much the same for nonclustered indexes as it does for clustered indexes, but there are some differences. In nonclustered indexes:

- Leaf pages are not the same as the data pages.
- Leaf level stores one key-pointer pair for each row in the table.
- Leaf-level pages store the index keys, data page number, and row number for the data row to which this index row is pointing. This combination of page number and row offset number is called the **row ID**.
- The root and intermediate levels store index keys and page pointers to other index pages. They also store the row ID of the key's data row.

With keys of the same size, nonclustered indexes require more space than clustered indexes.

Leaf pages revisited

The leaf page of an index is the lowest level of the index where all of the keys for the index appear in sorted order.

In clustered indexes on allpages-locked tables, the data rows are stored in order by the index keys, so by definition, the data level is the leaf level. There is no other level of the clustered index that contains one index row for each data row. Clustered indexes on allpages-locked tables are sparse indexes.

The level above the data contains one pointer for every data page, not data row.

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

Nonclustered index structure

The table in Figure 5-7 shows a nonclustered index on `lname`. The data rows at the far right show pages in ascending order by `employee_id` (10, 11, 12, and so on) because there is a clustered index on that column.

The root and intermediate pages store:

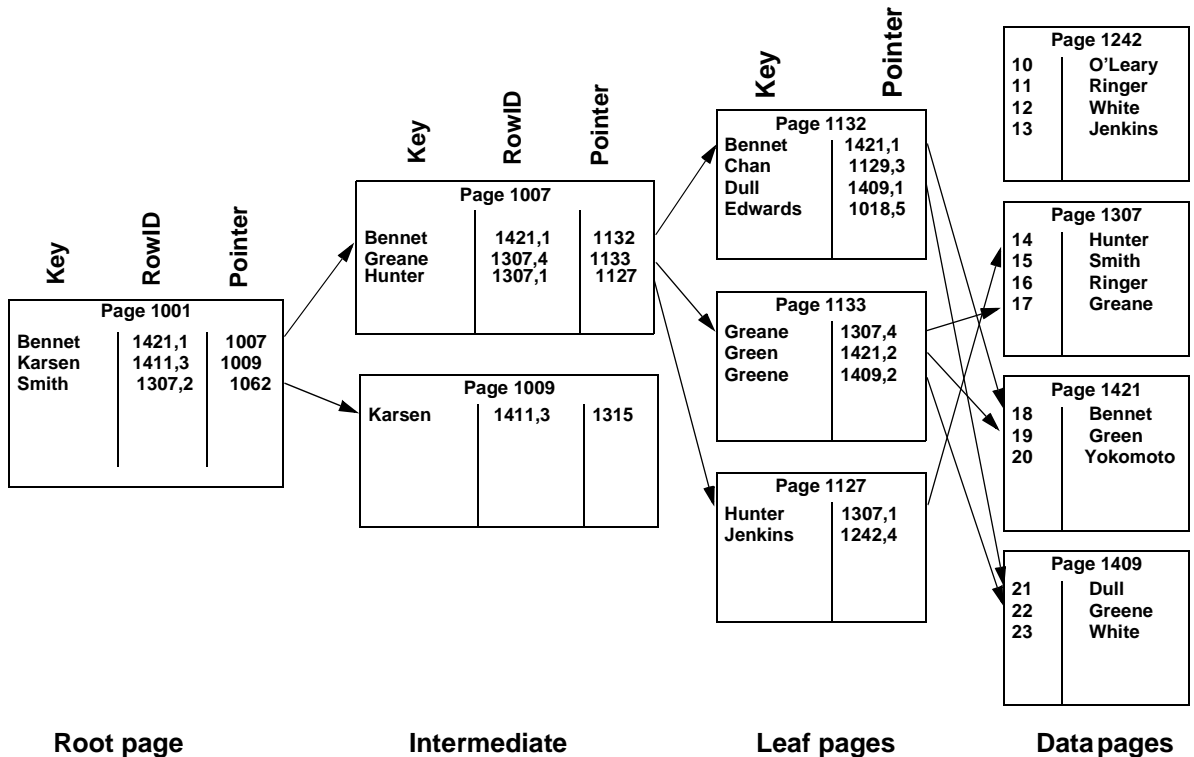
- The key value
- The row ID
- The pointer to the next level of the index

The leaf level stores:

- The key value
- The row ID

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

Figure 5-7: Nonclustered index structure



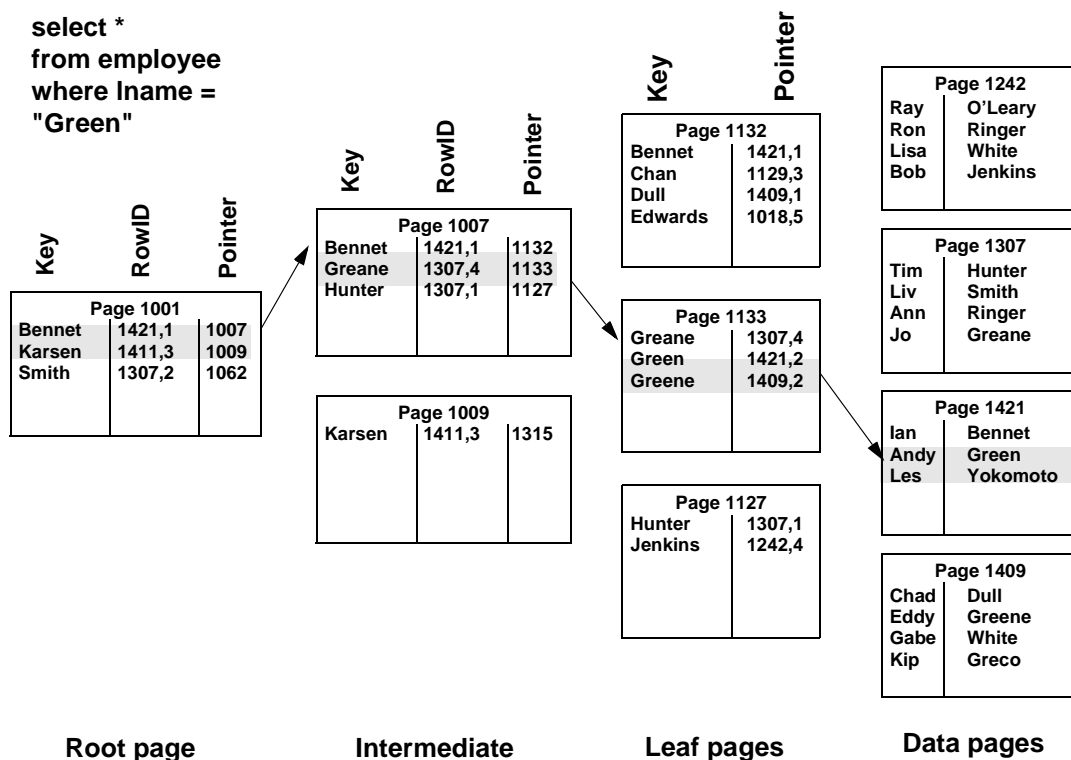
Nonclustered indexes and select operations

When you select a row using a nonclustered index, the search starts at the root level. `syspartitions` stores the page number for the root page of the nonclustered index (stored in `sysindexes` in Adaptive Server versions earlier than 15.0).

In Figure 5-8, “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 5-8: Selecting rows using a nonclustered index



Nonclustered index performance

The query in Figure 5-8 requires one read for each:

- Root level page
- Intermediate level page
- Leaf-level page
- Data page

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

Nonclustered indexes and insert operations

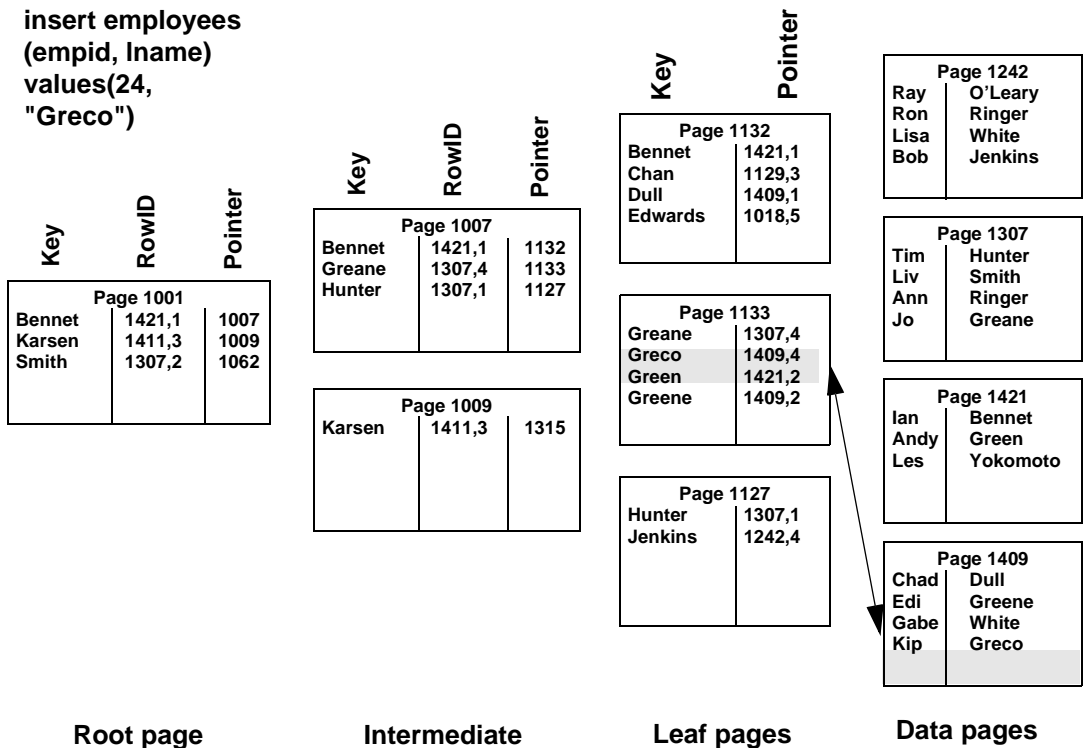
When you insert rows into a heap that has a nonclustered index and no clustered index, the rows are inserted in the last page of the table.

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, the clustered index 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 5-9 shows an insert into a heap table with a nonclustered index. The row is placed at the end of the table. A row containing the new key value and the row ID is also inserted into the leaf level of the nonclustered index.

Figure 5-9: An insert into a heap table with a nonclustered index

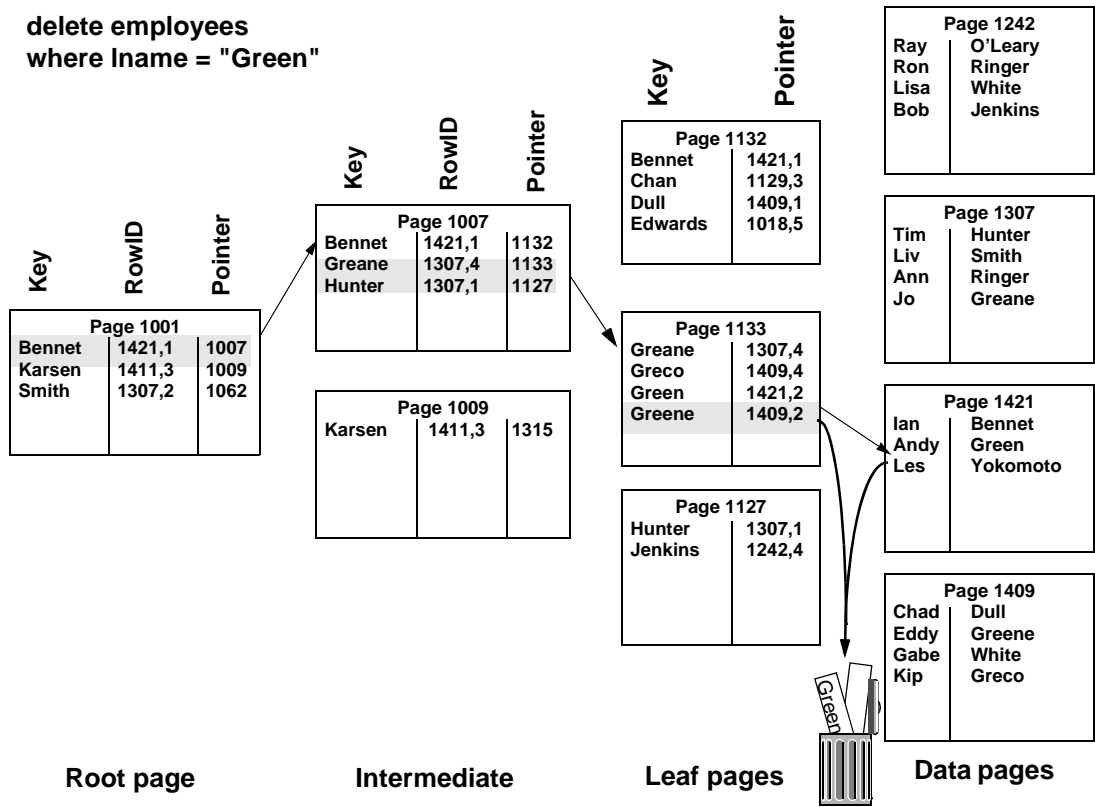


Nonclustered indexes and delete operations

When a row is deleted from a table, the query can use a nonclustered index on the columns in the `where` clause to locate the data row to delete, as shown in Figure 5-10.

The row in the leaf level of the nonclustered index that points to the data row is also removed. If there are other nonclustered indexes on the table, the rows on the leaf level of those indexes are also deleted.

Figure 5-10: Deleting a row from a table with a nonclustered index



If the deletion 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 105.

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 is 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 “Setting Space Management Properties,” in *Performance and Tuning Series:Physical Database Tuning*.

Index covering

Index covering can produce dramatic performance improvements when all columns needed by the query are included in the index.

You can create indexes on more than one key. These are called composite indexes. Composite indexes can have up to 31 columns, adding up to a maximum 600 bytes.

If you create a composite nonclustered index on each column referenced in the query's `select` list and in any `where`, `having`, `group by`, and `order by` clauses, the query can be satisfied by accessing only the index.

Since the leaf level of a nonclustered index or a clustered index on a data-only-locked table contains the key values for each row in a table, queries that access only the key values can retrieve the information by using the leaf level of the nonclustered index as if it were the actual table data. This is called index covering.

Both matching and nonmatching index scans can use an index that covers a query.

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 134 describes query types that make good use of covering indexes.

Covering matching index scans

Covering matching index scans lets you skip the last read for each row returned by the query, the read that fetches the data page.

For point queries that return only a single row, the query's performance gain is slight—just one page.

For range queries, the performance gain is larger, since the covering index saves one read for each row returned by the query.

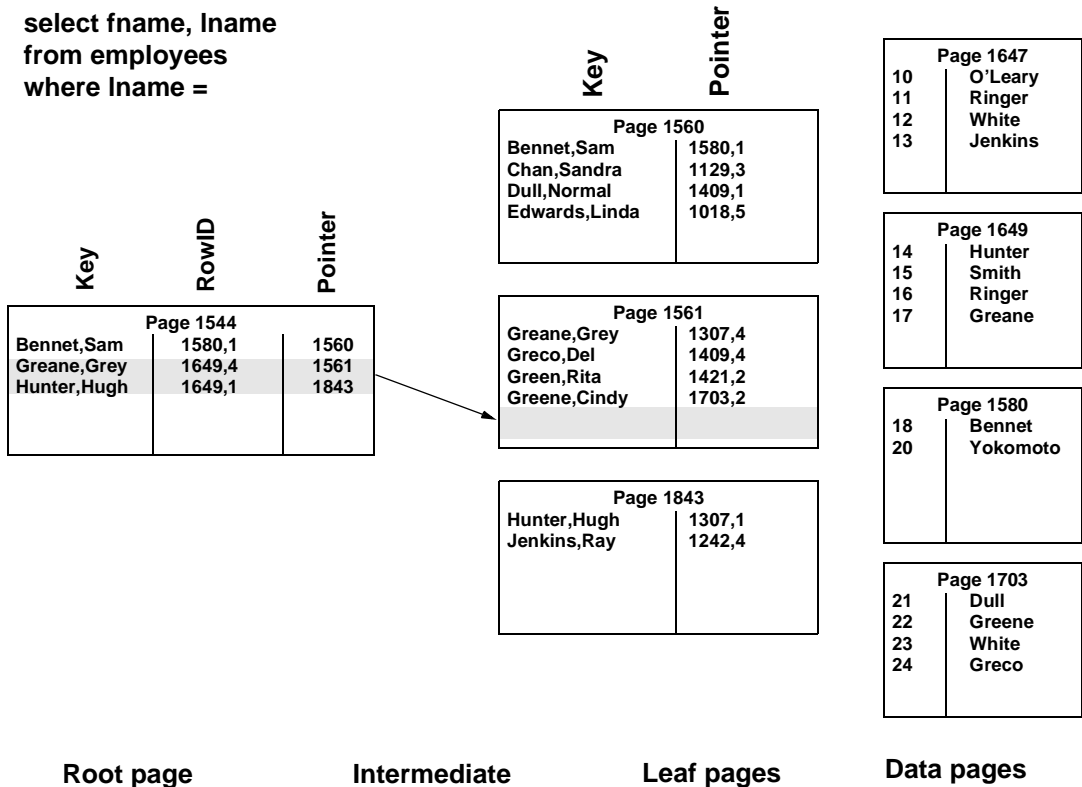
For a covering matching index scan to be used, the index must contain all columns named in the query. In addition, the columns in the `where` clauses of the query must include the leading column of the columns in the index.

For example, for an index on columns A, B, C, and D, the following sets can perform matching scans: A, AB, ABC, AC, ACD, ABD, AD, and ABCD. The columns B, BC, BCD, BD, C, CD, or D do not include the leading column and can be used only for nonmatching scans.

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

In Figure 5-11, the nonclustered index on `lname`, `fname` covers the query. The `where` clause includes the leading column, and all columns in the `select` list are included in the index, so the data page need not be accessed.

Figure 5-11: Matching index access does not have to read the data row



Covering nonmatching index scans

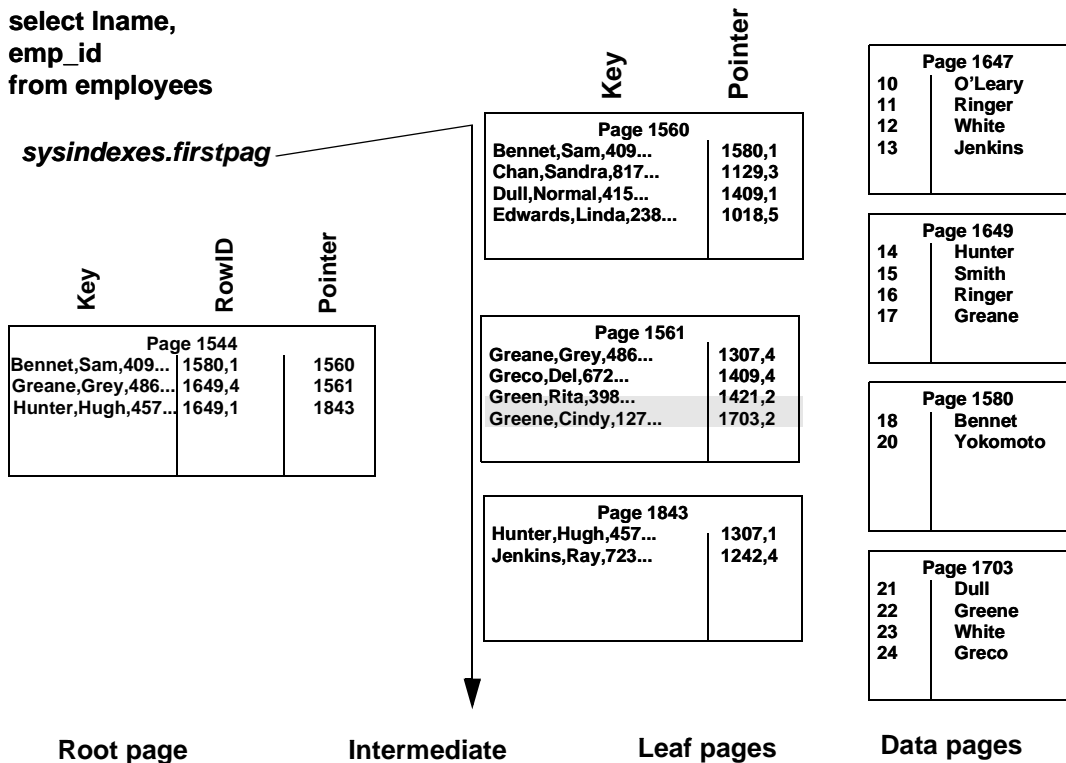
When the columns specified in the *where* clause do not include the leading column in the index, but all columns named in the select list and other query clauses (such as *group by* or *having*) are included in the index, Adaptive Server saves I/O by scanning the entire leaf level of the index, rather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

It cannot perform a matching scan because the first column of the index is not specified.

The query in Figure 5-12 shows a nonmatching index scan. This query does not use the leading columns on the index, but all columns required in the query are in the nonclustered index on *lname*, *fname*, *emp_id*.

The nonmatching scan must examine all rows on the leaf level. It scans all leaf level index pages, starting from the first page. It has no way of knowing how many rows might match the query conditions, so it must examine every row in the index. Since it must begin at the first page of the leaf level, it can use the pointer in `syspartitions.firstpage` rather than descend the index.

Figure 5-12: A nonmatching index scan



Indexes and caching

“How Adaptive Server performs I/O for heap operations” in *Performance and Tuning Series: Physical Database Tuning* 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:

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

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

Each time a page is found in cache, it is moved to the MRU end of the page chain, so the root page and higher levels of the index tend to stay in the cache.

Using separate caches for data and index pages

Indexes and the tables they index can use different caches. A system administrator or table owner can bind a clustered or nonclustered index to one cache and its table to another.

Index trips through the cache

A special strategy keeps index pages in cache. Data pages make only a single trip through the cache: they are read in at the MRU end of the cache or placed immediately before the **wash marker** (A point in the cache on the MRU/LRU chain), depending on the cache strategy chosen for the query.

Once the pages reach the LRU end of the cache, the buffer for that page is reused when another page needs to be read into cache.

For index pages, a counter controls the number of trips that an index page can make through the cache.

When the counter is greater than 0 for an index page, and it reaches the LRU end of the page chain, the counter is decremented by 1, and the page is placed at the MRU end again.

By default, the number of trips that an index page makes through the cache is set to 0. To change the default, a system administrator can set the `number of index trips` configuration parameter.

Indexing for Concurrency Control

This chapter introduces the basic query analysis tools that can help you choose appropriate indexes. It also discusses index selection criteria for point queries, range queries, and joins.

| Topic | Page |
|----------------------------------|------|
| How indexes affect performance | 117 |
| Symptoms of poor indexing | 119 |
| Detecting indexing problems | 119 |
| Fixing corrupted indexes | 122 |
| Index limits and requirements | 125 |
| Choosing indexes | 125 |
| Techniques for choosing indexes | 137 |
| Index and statistics maintenance | 140 |
| Additional indexing tips | 141 |

How indexes affect performance

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

Analyze your application workload and create indexes as necessary to improve the performance of the most critical processes.

The Adaptive Server query optimizer 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 sorting data or to favor the less costly ordered-input based JOIN, UNION, GROUP, or DISTINCT operators over other more expensive algorithms (for example, using merge joins instead of nested-loop joins and so on).

For example, to select the best index for a join clause:

```
r.c1=s.c1 and ... r.cn=s.cn
```

- Indexes on *r* or *s* that have any subset of *c1* ... *cn* as a prefix avoid the sort on the side of the merge join with the prefix.
- You can use indexes on both sides of the *and* clause if they are compatible (that is, they have a nonempty common prefix covered by the *equijoin* clause. This common prefix determines the part of the *equijoin* clause used as a merge clause (the longer the merge clause, the more effective it is).
- The query processor enumerates plans with an index on one side and a sort on the other. In the example above, the index prefix covered by the *equijoin* clause determines the part of the *equijoin* clause used as a merge clause (again, the longer the merge clause, the more effective it is).

You can use similar steps to identify the best index for union, distinct, and group clauses.

You can create indexes to enforce the uniqueness of data and to randomize the storage location of inserts.

You can set `sp_chgattribute 'concurrency_opt_threshold'` parameter to avoid table scans for increased concurrency. The syntax is:

```
sp_chgattribute table_name, "concurrency_opt_threshold", min_page_count
```

For example, this sets the concurrency optimization threshold for a table to 30 pages:

```
sp_chgattribute lookup_table, "concurrency_opt_threshold", 30
```

Detecting indexing problems

Some of the major indications of insufficient or incorrect indexing include:

- A `select` statement takes too long.
- A join between two or more tables takes an extremely long time.
- `select` operations perform well, but data modification processes perform poorly.
- Point queries (for example, `where colvalue = 3`) perform well, but range queries (for example, `where colvalue > 3 and colvalue < 30`) perform poorly.

These underlying problems are described in the following sections.

Symptoms of poor indexing

A primary goal of improving performance with indexes is avoiding table scans (which read every page of the table from disk), or partial table scans, which read only data pages from disk.

A query searching for a unique value in a table that has 600 data pages requires 600 physical and logical reads. If an index points to the data value, the same query can be satisfied with 2 or 3 reads, a 200 to 300 fold performance improvement

On a system with a 12-millisecond disk, this is a difference of several seconds compared to less than a second. Heavy disk I/O by a single query has a negative impact on overall throughput.

Lack of indexes is causing table scans

If `select` operations and joins take too long, it probably indicates that either an appropriate index does not exist or, it exists, but is not being used by the optimizer.

`showplan` output reports whether the table is being accessed by a table scan or index. If you think an index should be used, but `showplan` reports a table scan, `dbcc traceon(302)` output can help you determine the reason. `dbcc traceon` displays the costing computations for all optimizing query clauses.

If there is no clause is included in `dbcc traceon(302)` output, there may be problems with the way the clause is written. If a clause that you think should limit the scan is included in `dbcc traceon(302)` output, look carefully at its costing, and that of the chosen plan reported with `dbcc traceon(310)`. See *Adaptive Server Reference Manual: Commands* for more information about `dbcc traceon`.

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 passport 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 uses such an index only in very special cases.

Index does not support range queries

Generally, clustered indexes and covering indexes provide good performance for range queries and for search arguments 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 an index's total key length up to one-third the page size. However, indexes with this key length can store very few rows per index page, and the index level may be high. This increases the number of pages to traverse from the index root to the leaf pages, and increases the amount of disk I/O needed during queries.

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 on a server configured for 2K pages.

```
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 6-1 shows the results.

Table 6-1: Effects of key size on index size and levels

| Index, key size | Leaf-level pages | Index levels |
|-----------------|------------------|--------------|
| t10, 10 bytes | 4311 | 3 |
| t20, 20 bytes | 6946 | 3 |
| t40, 40 bytes | 12501 | 4 |

The output shows that the indexes for the 10-column and 20-column keys each have three levels, while the 40-column key requires a fourth level.

The number of pages required is more than 50 percent higher at each level.

Exception for wide data rows and wide index rows

Indexes with wide rows may be useful when:

- The table has very wide rows, resulting in very few rows per data page.
- The set of queries run on the table provides logical choices for a covering index.
- Queries return a sufficiently large number of rows.

For example, if a table has very long rows, and only one row per page, a query that needs to return 100 rows must access 100 data pages. An index that covers this query, even with long index rows, can improve performance.

For example, if the index rows are 240 bytes, the index stores 8 rows per page, and the query must access only 12 index pages.

Fixing corrupted indexes

If the index on one of your system tables has been corrupted, you can use the `sp_fixindex` system procedure to repair the index. For syntax information, see the entry for `sp_fixindex` in “System Procedures” in the *Adaptive Server Reference Manual*.

v Repairing the system table index with `sp_fixindex`

- 1 Get the *object_name*, *object_ID*, and *index_ID* of the corrupted index. If you only have a page number and you need to find the *object_name*, see the *Adaptive Server Troubleshooting and Error Messages Guide* for instructions.
- 2 If the corrupted index is on a system table in the master database, put Adaptive Server in single-user mode. See the *Adaptive Server Troubleshooting and Error Messages Guide* for instructions.
- 3 If the corrupted index is on a system table in a user database, put the database in single-user mode and reconfigure to allow updates to system tables:

```
1> use master
2> go
1> sp_dboption database_name, "single user", true
2> go
1> sp_configure "allow updates", 1
2> go
```

- 4 Issue the `sp_fixindex` command:

```
1> use database_name
2> go

1> checkpoint
2> go

1> sp_fixindex database_name, object_name, index_ID
```

```
2> go
```

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

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

Note You must be assigned `sa_role` to run `sp_fixindex`.

5 Run `dbcc checktable` to verify that the corrupted index is now fixed.

6 Disallow updates to system tables:

```
1> use master
```

```
2> go
```

```
1> sp_configure "allow updates", 0
```

```
2> go
```

7 Turn off single-user mode:

```
1> sp_dboption database_name, "single user", false
```

```
2> go
```

```
1> use database_name
```

```
2> go
```

```
1> checkpoint
```

```
2> go
```

You can use the `checkpoint` to identify the one or more databases or use an `all` clause, which means you do not have to issue the `use database` command.

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

v **Repairing a nonclustered index on *sysobjects***

1 Perform steps 1 – 3, as described in “Repairing the system table index with `sp_fixindex`,” above.

2 Issue:

```
1> use database_name
```

```
2> go
```

```
1> checkpoint
```

```
2> go
```

```
1> select sysstat from sysobjects
```

```
2> where id = 1
```

```
3> go
```

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

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

3 Save the original sysstat value.

4 Change the sysstat column to the value required by sp_fixindex:

```
1> update sysobjects
2> set sysstat = sysstat | 4096
3> where id = 1
4> go
```

5 Run:

```
1> sp_fixindex database_name, sysobjects, 2
2> go
```

6 Restore the original sysstat value:

```
1> update sysobjects
2> set sysstat = sysstat_ORIGINAL
3> where id = object_ID
4> go
```

7 Run dbcc checktable to verify that the corrupted index is now fixed.

8 Disallow updates to system tables:

```
1> sp_configure "allow updates", 0
2> go
```

9 Turn off single-user mode:

```
1> sp_dboption database_name, "single user", false
2> go

1> use database_name
2> go

1> checkpoint
2> go
```

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

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

Index limits and requirements

These limits apply to indexes in Adaptive Server:

- Because the data for a cluster index is ordered by index key, you can create only one clustered index per table. Adaptive Server creates a clustered index by default as a local index for range-, list-, and hash-partitioned tables. You cannot create global clustered indexes on range-, list-, or hash-partitioned tables.
- You can create a maximum of 249 nonclustered indexes per table.
- When you create a clustered index, Adaptive Server requires empty free space to copy the rows in the table and allocate space for the clustered index pages. It also requires space to re-create any nonclustered indexes on the table.

The amount of space required can vary, depending on how full the table's pages are when you begin and the space management properties are applied to the table and index pages.

See “Determining the space available for maintenance activities” in “Database Maintenance,” in *Performance and Tuning Series: Physical Database Tuning*.

- 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.
- A key can be made up of as many as 31 columns. The maximum number of bytes per index key is varies by the page size in bytes as follows:

| Page Size | Max key length |
|-----------|----------------|
| 2048 | 600 |
| 4096 | 1250 |
| 8192 | 2600 |
| 16384 | 5300 |

Choosing indexes

When you are working with index selection you may want to ask these questions:

- What indexes are associated currently with a given table?
- What are the most important processes that make use of the table?
- What is the ratio of `select` operations to data modifications performed on the table?
- Has a clustered index been created for the table?
- Can the clustered index be replaced by a nonclustered index?
- Do any of the indexes cover one or more of the critical queries?
- Is a composite index required to enforce the uniqueness of a compound primary key?
- Do existing queries contain expressions that could be accelerated by using function-based indexes?
- What indexes can be defined as unique?
- What are the major sorting requirements?
- Do some queries use descending ordering of result sets?
- Do the indexes support joins and referential integrity checks?
- Does indexing affect update types (direct versus deferred)?
- What indexes are needed for cursor positioning?
- If dirty reads are required, are there unique indexes to support the scan?
- Should `IDENTITY` columns be added to tables and indexes to generate unique indexes? Unique indexes are required for updatable cursors and dirty reads.

When deciding how many indexes to use, consider:

- Space constraints
- Access paths to table
- Percentage of data modifications versus select operations
- Performance requirements of reports versus OLTP
- Performance impacts of index changes
- How often you can use update statistics

Index keys and logical keys

You must differentiate index keys 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 heap tables.

In a high-transaction environment, the locking on the last page severely limits throughput.

- If your environment requires a lot of inserts, do not place the clustered index key on a steadily increasing value such as an IDENTITY column. Instead, 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 often 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 performed 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 change infrequently
- 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 not be supported by nonclustered indexes, depending on the size of the range.

Consider using composite indexes to cover critical queries and to support less frequent queries:

- The most critical queries should be able to perform point queries and matching scans.
- Other queries should be able to perform nonmatching scans using the index, which avoids table scans.

Choosing function-based indexes

Function-based indexes can provide an inexpensive option for enhancing the performance of certain legacy applications.

Function-based indexes allow you to create indexes based directly on one or more expressions (see the *Transact-SQL User's Guide*). When the index is built, the result of evaluating the expressions for each row is stored as an index key value, and is not reevaluated at query execution time. This means lookups on the result of an expression within a SQL query can be very fast. Without function-based indexes, table scans are typically required to evaluate the expression for each row in the table for comparison. Adaptive Server creates a hidden computed column containing the evaluated key expressions and indexes this column.

You can effectively use function-based indexes for queries that need to apply a function or operation to a column value and compare the result to another column in the same row or to a constant or variable.

You can also obtain the performance benefits of function-based indexes by adding a materialized computed column with index to a table and rewriting the query to use the indexed computed column. This can be a good approach for new application development. The advantage of function-based indexes is that you can simply add to an existing table an index that matches expressions used in existing queries. In this way, you can enhance the performance of legacy applications with a minimal schema addition and no change to SQL query code.

Index selection

Index selection allows you to determine which indexes are actively being used and those that are rarely used.

This section assumes that the monitoring tables feature is already set up. See the *Performance and Tuning Series: Monitoring Tables* for information about installing and using the monitoring tables.

Index selection uses these columns of the monitoring access table, `monOpenObjectActivity`:

- `IndexID` – unique identifier for the index.
- `OptSelectCount` – reports the number of times that the corresponding object (such as a table or index) was used as the access method by the optimizer.
- `LastOptSelectDate` – reports the last time `OptSelectCount` was incremented.
- `UsedCount` – reports the number of times that the corresponding object (such as a table or index) was used as an access method when a query executed.
- `LastUsedDate` – reports the last time `UsedCount` was incremented.

If a plan has already been compiled and cached, `OptSelectCount` is not incremented each time the plan is executed. However, `UsedCount` is incremented when a plan is executed. If `no_exec` is on, `OptSelectCount` is incremented, but `UsedCount` is not.

Monitoring data is nonpersistent. That is, when you restart the server, the monitoring data is reset. Monitoring data is reported only for active objects. For example, monitoring data does not exist for objects that have not been opened, since there are no active object descriptors for such objects. If the system is inadequately configured and has reused object descriptors, monitoring data for these object descriptors is reinitialized and the data for the previous object is lost. When the old object is reopened, its monitoring data is reset.

Examples of using index selection

The following example queries the monitoring tables for the last time all indexes for a specific object were selected by the optimizer, as well as the last time they were actually used during execution, and reports the counts in each case:

```
select DBID, ObjectID, IndexID, OptSelectCount, LastOptSelectDate, UsedCount,
LastUsedDate
from monOpenObjectActivity
where DBID=db_id("financials_db") and ObjectID =
object_id('financials_db..expenses')
```

order by UsedCount

This example displays all indexes that are used—or not currently used—in an application:

```
select DBID, ObjectID, IndexID, ObjectName = object_name(ObjectID, DBID),
LastOptSelectDate, UsedCount, LastUsedDate
from monOpenObjectActivity
where DBID = db_id("MY_1253_RS_RSSD")
and ObjectID = object_id('MY_1253_RS_RSSD..rs_columns')
```

| DBID | ObjectID | IndexID | ObjectName |
|--------------------|-----------|-----------|--------------------|
| LastOptSelectDate | | UsedCount | LastUsedDate |
| ----- | ----- | ----- | ----- |
| 4 | 192000684 | 0 | rs_columns |
| May 15 2006 4:18PM | | 450 | May 15 2006 4:18PM |
| 4 | 192000684 | 1 | rs_columns |
| NULL | | 0 | NULL |
| 4 | 192000684 | 2 | rs_columns |
| NULL | | 0 | NULL |
| 4 | 192000684 | 3 | rs_columns |
| May 12 2006 6:11PM | | 1 | May 12 2006 6:11PM |
| 4 | 192000684 | 4 | rs_columns |
| NULL | | 0 | NULL |
| 4 | 192000684 | 5 | rs_columns |
| NULL | | 0 | NULL |

If the index is not used, it results in a NULL date. If an index is used, it results in a date like “May 15 2006 4:18PM.”

In this example, the query displays all indexes that are not currently used in the current database:

```
select DB = convert(char(20), db_name()),
TableName = convert(char(20), object_name(i.id, db_id())),
IndexName = convert(char(20), i.name),
IndID = i.indid
from master..monOpenObjectActivity a, sysindexes i
where a.ObjectID =* i.id
and a.IndexID =* i.indid
and (a.UsedCount = 0 or a.UsedCount is NULL)
and i.indid > 0
and object_name(i.id, db_id()) not like "sys%"
order by 2, 4 asc
```

| DB | TableName | IndexName | IndID |
|-----------------|-------------|------------------|-------|
| ----- | ----- | ----- | ----- |
| MY_1253_RS_RSSD | rs_articles | rs_key_articles | 1 |
| MY_1253_RS_RSSD | rs_articles | rs_key4_articles | 2 |

| | | | |
|-----------------|--------------|--------------------|---|
| MY_1253_RS_RSSD | rs_classes | rs_key_classes | 1 |
| MY_1253_RS_RSSD | rs_classes | rs_key2_classes | 2 |
| MY_1253_RS_RSSD | rs_config | rs_key_config | 1 |
| MY_1253_RS_RSSD | rs_databases | rs_key_databases | 1 |
| MY_1253_RS_RSSD | rs_databases | rs_key9_databases | 2 |
| MY_1253_RS_RSSD | rs_databases | rs_key13_databases | 3 |
| MY_1253_RS_RSSD | rs_databases | rs_key14_databases | 4 |
| MY_1253_RS_RSSD | rs_databases | rs_key15_databases | 5 |
| MY_1253_RS_RSSD | rs_datatype | rs_key_datatypes | 1 |
| MY_1253_RS_RSSD | rs_datatype | rs_key2_datatype | 2 |

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” in “Optimization for Cursors” in *Performance and Tuning Series: Query Processing and Abstract Plans*.
- If you are creating an index on a table that will have a lot of insert activity, use `fillfactor` to temporarily minimize page splits, 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.
 - Internally, numerics compare slightly faster than strings.

- 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, by definition, variable-length.
- Whenever possible, use fixed-length, nonnull 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.

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, consider:

- 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

For all locking schemes, each nonclustered index needs to be updated for each insertion into, and each deletion from, the table

An update to the table that changes part of an index's key requires only that index be updated.

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
- 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, provide indexes for all frequently used 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 nonleading 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 6-2 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 6-2: Composite nonclustered index ordering and performance

| Columns in the <code>where</code> clause | Performance with the indexed columns in the select list | Performance with other columns in the select list |
|--|--|---|
| <code>au_lname</code> or <code>au_lname, au_fname</code> or <code>au_lname, au_fname, au_id</code> | Good; index used to descend tree; data level is not accessed | Good; index used to descend tree; data is accessed (one more page read per row) |
| <code>au_fname</code> or <code>au_id</code> or <code>au_fname, au_id</code> | Moderate; index is scanned to return values | Poor; index not used, table scan |

Choose the ordering of the composite index so that most queries form a prefix subset.

Advantages and disadvantages of composite indexes

Composite indexes have these advantages:

- A composite index provides opportunities for index covering.
- If queries provide search arguments on each of the keys, the composite index requires fewer I/Os than the same query using an index on any single attribute.
- A composite index is a good way to enforce the uniqueness of multiple attributes.

Good choices for composite indexes are:

- Lookup tables
- Columns that are frequently accessed together
- Columns used for vector aggregates
- Columns that make a frequently used subset from a table with very wide rows

The disadvantages of composite indexes are:

- Composite indexes tend to have large entries. This means fewer index entries per index page and more index pages to read.
- An update to any attribute of a composite index causes the index to be modified. The columns you choose should not be those that are updated often.

Poor choices are:

- Index rows that are nearly as wide as the data rows
- Composite indexes where only a minor key is used in the `where` clause

Using online reorg rebuild for data-only-locked indexes

You can run `online reorg rebuild index` on DOL indexes to recompact the data, collect garbage from spaces left by previous deallocations, and rearrange the data to improve the index page clustering ratio. Running `online reorg rebuild index` reduces the space an index requires, and improves the query execution with higher clustering.

Techniques for choosing indexes

This section presents a study of two queries that must access a single table, and the indexing choices for these two queries. The two queries are:

- A range query that returns a large number of rows
- A point query that returns only one or two rows

Choosing an index for a range query

Assume that you need to improve the performance of the following query:

```
select title
from titles
where price between $20.00 and $30.00
```

Some basic statistics on the table are:

- The table has 1,000,000 rows, and uses allpages locking.
- There are 10 rows per page; pages are 75 percent full, so the table has approximately 135,000 pages.
- 190,000 (19%) of the titles are priced between \$20 and \$30.

With no index, the query would scan all 135,000 pages.

With a clustered index on `price`, the query would find the first \$20 book and begin reading sequentially until it gets to the last \$30 book. With pages about 75 percent full, the average number of rows per page is 7.5. To read 190,000 matching rows, the query would read approximately 25,300 pages, plus 3 or 4 index pages.

With a nonclustered index on `price` and random distribution of `price` values, using the index to find the rows for this query requires reading about 19 percent of the leaf level of the index; about 1,500 pages.

If the `price` values are randomly distributed, the number of data pages that must be read is likely to be high, perhaps as many data pages as there are qualifying rows, 190,000. Since a table scan requires only 135,000 pages, you would not want to use a nonclustered index.

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 6-3 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 6-3: Comparing index strategies for two queries

| Possible index choice | Index pages | Range query on price | Point query on title |
|-------------------------|-------------|-------------------------------|----------------------------|
| 1 Nonclustered on title | 36,800 | Clustered index, about 26,600 | Nonclustered index, 6 I/Os |
| Clustered on price | 650 | pages (135,000 *.19) | |
| | | With 16K I/O: 3,125 I/Os | |

| Possible index choice | Index pages | Range query on price | Point query on title |
|---|----------------|---|---|
| 2 Clustered on title Nonclustered on price | 3,770 6,076 | Table scan, 135,000 pages With 16K I/O: 17,500 I/Os | Clustered index, 6 I/Os |
| 3 Nonclustered on title, price | 36,835 | Nonmatching index scan, about 35,700 pages With 16K I/O: 4,500 I/Os | Nonclustered index, 5 I/Os |
| 4 Nonclustered on price, title | 36,835 | Matching index scan, about 6,800 pages (35,700 *.19) With 16K I/O: 850 I/Os | Nonmatching index scan, about 35,700 pages With 16K I/O: 4,500 I/Os |

Examining the figures in Table 6-3 shows that:

- For the range query on price, choice 4 is best; choices 1 and 3 are acceptable with 16K I/O.
- For the point query on titles, indexing choices 1, 2, and 3 are excellent.

The best indexing strategy for a combination of these two queries is to use two indexes:

- Choice 4, for range queries on price.
- Choice 2, for point queries on title, since the clustered index requires very little space.

You may need additional information to help you determine which indexing strategy to use to support multiple queries. Typical considerations are:

- What is the frequency of each query? How many times per day or per hour is the query run?
- What are the response time requirements? Is one of them especially time critical?
- What are the response time requirements for updates? Does creating more than one index slow updates?
- Is the range of values typical? Is a wider or narrower range of prices, such as \$20 to \$50, often used? How do different ranges affect index choice?
- Is there a large data cache? Are these queries critical enough to provide a 35,000-page cache for the nonclustered composite indexes in index choice 3 or 4? Binding this index to its own cache would provide very fast performance.
- What other queries and what other search arguments are used? Is this table frequently joined with other tables?

Index and statistics maintenance

To ensure that indexes evolve with your system:

- Monitor queries to determine if indexes are still appropriate for your applications.

Periodically, check the query plans, as described in “Using showplan,” in *Performance and Tuning Series: Query Processing and Abstract Plans* and the I/O statistics for your most frequent user queries. Pay special attention to noncovering indexes that support range queries. They are most likely to switch to table scans if the data distribution changes

- Drop and rebuild indexes that hurt performance.
- Keep index statistics up to date.
- Use space management properties to reduce page splits and to reduce the frequency of maintenance operations.

Dropping indexes that hurt performance

Drop indexes that hurt performance. If an application performs data modifications during the day and generates reports at night, you may want to drop some indexes in the morning and re-create them at night.

Many system designers create numerous indexes that are rarely, if ever, actually used by the query optimizer. Make sure that you base indexes on the current transactions and processes that are being run, not on the original database design.

Check query plans to determine whether your indexes are being used.

See “Maintaining index and column statistics” and “Rebuilding indexes” in “Maintenance Activities and Performance,” in *Performance and Tuning Series: Physical Database Tuning*.

Choosing space management properties for indexes

Space management properties can help reduce the frequency of index maintenance. In particular, choosing the fillfactor value 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 “Setting Space Management Properties,” in *Performance and Tuning Series: Physical Database Tuning*.

Additional indexing tips

These suggestions may improve 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 perform frequent index maintenance, configure your server to speed up the sorting.

See “Configuring Adaptive Server to speed sorting” in “Maintenance Activities and Performance,” in *Performance and Tuning Series: Physical Database Tuning* for information about configuration parameters that enable faster sorting.

Creating artificial columns

When indexes, especially composite indexes, become too large, it may be 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. This speeds the inserts and bulk copies, since the nonclustered indexes do not have to be updated with every insert.

See “Rebuilding indexes” in “Database Maintenance” in *Performance and Tuning Series: Physical Database Tuning*.

Configuring enough sort buffers

The sort buffers decide how many pages of data you can sort in each run. The number of pages is the basis for the logarithmic function used to calculate the number of runs needed to finish the sort.

For example, if you have 500 buffers, then the number of runs is calculated with “log (number of pages in table) with 500 as the log base.”

Also, the number of sort buffers is shared by threads in the parallel sort; if you do not have enough sort buffers, the parallel sort may not work as fast as it should.

Creating the clustered index first

Do not create nonclustered indexes, then clustered indexes. When you create the clustered index, all previous nonclustered indexes are rebuilt.

Configure large buffer pools

To set up for larger I/Os, configure large buffers pools in a named cache and bind the cache to the table.

Asynchronous log service

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

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

You can enable, disable, or configure ALS using `sp_dboption`:

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

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

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

You can use the `checkpoint` to identify one or more databases, or use an `all` clause.

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

Disabling ALS

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

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

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

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

Displaying ALS

You can see whether ALS is enabled in a specified database using:

```
-----  
sp_helpdb "mydb"  
-----  
mydb          3.0 MB sa          2  
              July 09, 2002  
              select into/bulkcopy/pllsort, trunc log on chkpt,  
              async log service
```

Understanding the user log cache (ULC) architecture

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

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

- 1 Obtain a lock on the last log page.
- 2 Allocate new log pages if necessary.
- 3 Copy the log records from the ULC to the log cache.

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

- 4 Flush the log cache to disk.

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

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

When to use ALS

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

- Heavy contention on the last log page.

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

Table 6-4: Log page under contention

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

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

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

Table 6-5:

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

- Under utilized bandwidth in the log device.

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

Using ALS

Two threads—the ULC flusher and the log writer—scan the dirty buffers (buffers full of data not yet written to the disk), copy the data, and write it to the log.

ULC flusher

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

Log writer

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

Stored procedure support for ALS

`sp_dboption` and `sp_help` support asynchronous log service by:

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

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

Index

Symbols

- ::= (BNF notation)
 - in SQL statements xiii
- , (comma)
 - in SQL statements xiii
- { } (curly braces)
 - in SQL statements xiii
- () (parentheses)
 - in SQL statements xiii
- [] (square brackets)
 - in SQL statements xiii

A

- allpages locking 4
 - changing to with **alter table** 78
 - or strategy 31
 - specifying with **create table** 78
 - specifying with **select into** 81
 - specifying with **sp_configure** 77
- ALS
 - log writer 146
 - user log cache 144
 - when to use 145
- ALS, see asynchronous log service 143
- alter table** command
 - changing table locking scheme with 78–82
 - sp_dboption** and changing lock scheme 80
- alternative predicates
 - nonqualifying rows 33
- application design
 - deadlock avoidance 72
 - deadlock detection in 68
 - delaying deadlock checking 72
 - isolation level 0 considerations 21
 - levels of locking 43
 - primary keys and 132
 - user interaction in transactions 41

artificial columns 141

B

- Backus Naur Form (BNF) notation xiii
- batch processing
 - transactions and lock contention 41
- blocking 53
- blocking process
 - avoiding during mass operations 43
 - sp_lock** report on 64
 - sp_who** report on 61
- BNF notation in SQL statements xiii
- brackets. *See* square brackets []
- B-trees, index
 - nonclustered indexes 105

C

- case sensitivity
 - in SQL xiv
- chains of pages
 - overflow pages and 102
- clustered indexes 92
 - changing locking modes and 81
 - delete operations 103
 - guidelines for choosing 127
 - insert operations and 98
 - order of key values 96
 - overflow pages and 102
 - page reads 97
 - structure of 96
- column-level locking
 - pseudo- 34
- columns
 - artificial 141
- comma (.)
 - in SQL statements xiii

Index

- composite indexes 134
 - advantages of 136
- concurrency
 - deadlocks and 67
 - locking and 3, 67
- configuration (Server)
 - lock limit 44
- consistency
 - transactions and 2
- constraints
 - primary key** 125
 - unique 125
- contention
 - avoiding with clustered indexes 91
 - reducing 40
- contention, lock
 - locking scheme and 54
 - sp_object_stats** report on 74
- context* column of **sp_lock** output 64
- conventions
 - See also* syntax
 - Transact-SQL syntax xiii
 - used in the Reference Manual xiii
- CPU usage
 - deadlocks and 68
- create index** command
 - locks acquired by 29
- create table** command
 - locking scheme specification 78
- curly braces ({}) in SQL statements xiii
- cursors
 - close on endtran** option 88
 - isolation levels and 87
 - lock duration 28
 - lock type 28, 30
 - locking and 87–89
 - shared** keyword in 88

D

- data
 - consistency 2
 - uniqueness 91
- data modification
 - nonclustered indexes and 133

- number of indexes and 120
- data pages
 - clustered indexes and 96
 - full, and insert operations 99
- database design
 - indexing based on 140
 - logical keys and index keys 127
- databases
 - lock promotion thresholds for 44
- data-only locking (DOL) tables
 - maximum row size 79
 - or strategy and locking 31
- datapages locking
 - changing to with **alter table** 78
 - described 6
 - specifying with **create table** 78
 - specifying with **select into** 81
 - specifying with **sp_configure** 77
- datarows locking
 - changing to with **alter table** 78
 - described 7
 - specifying with **create table** 78
 - specifying with **select into** 81
 - specifying with **sp_configure** 77
- datatypes
 - choosing 132, 141
 - numeric compared to character 141
- deadlock checking period** configuration parameter 72
- deadlocks 67–73, 74
 - application-generated 67
 - avoiding 71
 - defined 67
 - delaying checking 72
 - detection 68, 74
 - diagnosing 53
 - error messages 68
 - performance and 39
 - read committed with lock** effects on 29
 - sp_object_stats** report on 74
 - worker process example 69
- delete**
 - clustered indexes 103
 - nonclustered indexes 110
 - transaction isolation levels and 23
 - uncommitted 32

demand locks 13
 sp_lock report on 64
 detecting deadlocks 74
 dirty reads 3
 preventing 21
 transaction isolation levels and 20
 duration of latches 18
 duration of locks
 read committed with lock and 29
 read-only cursors 30
 transaction isolation level and 26

E

error messages
 deadlocks 68
 escalation, lock 48
 exclusive locks
 page 9
 sp_lock report on 64
 table 10

F

family duration locks 64
 fetching cursors
 locking and 88
 fillfactor
 index creation and 132
 fixed-length columns
 for index keys 133
 overhead 133

H

holdlock keyword
 locking 85
 shared keyword and 88
 hot spots
 avoiding 42

I

IDENTITY columns
 indexing and performance 127
 index keys, logical keys and 127
 index pages
 locks on 5
 page splits for 101
 storage on 93
 index selection 129
 indexes 91–116
 access through 91
 design considerations 117
 dropping infrequently used 140
 guidelines for 132
 intermediate level 94
 leaf level 93
 leaf pages 105
 locking with 9
 number allowed 125
 partitions 95
 performance 91
 root level 93
 selectivity 120
 size of entries and performance 121
 types of 92
 indexing
 configure large buffer pools 143
 create a clustered index first 142
 infinity key locks 17
insert command
 contention and 42
 transaction isolation levels and 23
 insert operations
 clustered indexes 98
 nonclustered indexes 109
 page split exceptions and 100
 intent table locks 10
 sp_lock report on 64
 intermediate levels of indexes 94
 isolation levels 19–26, 82–87
 cursors 87
 default 82
 dirty reads 21
 lock duration and 26, 27, 28
 nonrepeatable reads 23
 phantoms 23

serializable reads and locks 17
 transactions 19

J

joins

choosing indexes for 128
 datatype compatibility in 133

K

key values

index storage 91
 order for clustered indexes 96
 overflow pages and 102

keys, index

choosing columns for 127
 clustered and nonclustered indexes and 92
 composite 134
 logical keys and 127
 monotonically increasing 101
 size and performance 132
 size of 125
 unique 132

L

latches 17

leaf levels of indexes 93

leaf pages 105

levels

indexes 93
 locking 43

lock allpages option

alter table command 79
create table command 78
select into command 81

lock datapages option

alter table command 79
create table command 78
select into command 81

lock datarows option

alter table command 79

create table command 78

select into command 81

lock duration. *See* duration of locks

lock promotion thresholds 44–52

database 51
 default 51
 dropping 52
 precedence 52
 promotion logic 50
 server-wide 51
 table 51

lock scheme configuration parameter 77

locking 1–45

allpages locking scheme 4
 commands 77–90
 concurrency 3
 contention, reducing 40–44
 control over 3, 8
 cursors and 87
 datapages locking scheme 6
 datarows locking scheme 7
 deadlocks 67–73
 entire table 8
for update clause 87
 forcing a write 13
holdlock keyword 83
 index pages 5
 indexes used 9
 isolation levels and 19–26, 82–87
 last page inserts and 127
 monitoring contention 56
 noholdlock keyword 83
noholdlock keyword 86
 overhead 3
 page and table, controlling 19, 47
 performance 39
read committed clause 84
read uncommitted clause 84, 86
 reducing contention 40
serializable clause 84
shared keyword 83, 86
sp_lock report on 63
 transactions and 3

locking scheme 53–57

allpages 4
 changing with **alter table** 78–82

- clustered indexes and changing 81
- create table** and 78
- datapages 6
- datarows 7
- lock types and 7
- server-wide default 77
- specifying with **create table** 78
- specifying with **select into** 81

locks

- blocking 61
- command type and 27, 28
- demand 13
- escalation 48
- exclusive page 9
- exclusive table 10
- family duration 64
- granularity 3
- infinity key 17
- intent table 10
- isolation levels and 27, 28
- latches and 17
- limits 29
- “lock sleep” status 61
- number of, data-only-locking 45
- or** queries and 31
- page 8
- reporting on 61
- shared page 9
- shared table 10
- size of 3
- table 10
- table versus page 48
- table versus row 48
- table, table scans and 30
- types of 7, 64
- update page 9
- viewing 63
- worker processes and 14

locktype column of **sp_lock** output 64

logical keys, index keys and 127

M

- matching index scans 112
- messages

- deadlock victim 68
- monitoring
 - index usage 140
 - indexes 129–132
 - indexes, examples of 130
 - lock contention 56
- multicolumn index. *See* composite indexes

N

- noholdlock** keyword, **select** 86
- nonclustered indexes 92
 - definition of 105
 - delete operations 110
 - guidelines for 128, 129
 - insert operations 109
 - number allowed 125
 - select** and 107
 - size of 105
 - structure 106
- nonmatching index scans 113–114
- nonrepeatable reads 23
- null columns
 - variable-length 132
- null values
 - datatypes allowing 132
- number (quantity of)
 - bytes per index key 125
 - clustered indexes 92
 - indexes per table 125
 - locks in the system 44
 - locks on a table 49
 - nonclustered indexes 92
- number of locks** configuration parameter
 - data-only-locked tables and 45
- number of sort buffers 142
- numbers
 - row offset 105

O

- observing deadlocks 74
- offset table
 - nonclustered index selects and 107

Index

- row IDs and 105
- optimistic index locking 57
 - added option in **sp_chgattribute** 57
 - cautions and issues 58
 - using 58
- optimizer
 - dropping indexes not used by 140
 - indexes and 117
 - nonunique entries and 120
- or** queries
 - allpages-locked tables and 31
 - data-only-locked tables and 31
 - isolation levels and 32
 - locking and 31
 - row requalification and 32
- order
 - composite indexes and 134
 - data and index storage 92
 - index key values 96
- order by** clause
 - indexes and 91
- output
 - sp_estspace** 121
- overflow pages 102
 - key values and 102
- overhead
 - datatypes and 132, 142
 - nonclustered indexes 133
 - variable-length columns 133

P

- page chains
 - overflow pages and 102
- page lock promotion HWM** configuration parameter 49
- page lock promotion LWM** configuration parameter 49
- page lock promotion PCT** configuration parameter 50
- page locks 7
 - sp_lock** report on 64
 - table locks versus 48
 - types of 8
- page splits
 - data pages 99
 - index pages and 101
 - nonclustered indexes, effect on 99

- performance impact of 101
- pages
 - overflow 102
- pages, data
 - splitting 99
- pages, index
 - leaf level 105
 - storage on 93
- parallel query processing
 - demand locks and 14
- parallel sort
 - configure enough sort buffers 142
- parentheses ()
 - in SQL statements xiii
- performance
 - clustered indexes and 56
 - data-only-locked tables and 56
 - indexes and 117
 - locking and 39
 - number of indexes and 120
- phantoms 16
 - serializable reads and 17
- phantoms in transactions 23
- pointers
 - index 93
- precedence
 - lock promotion thresholds 52
- primary key** constraint
 - index created by 125
- promotion, lock 48

Q

- qualifying old and new values
 - uncommitted updates 36
- queries
 - range 120

R

- range queries 120
- read committed with lock** configuration parameter
 - deadlocks and 29
 - lock duration 29

- reads
 - clustered indexes and 97
- reduce contention
 - suggestions 37
- referential integrity
 - references** and unique index requirements 132
- root level of indexes 93
- row ID (RID) 105
- row lock promotion HWM** configuration parameter 49
- row lock promotion LWM** configuration parameter 49
- row lock promotion PCT** configuration parameter 50
- row locks
 - sp_lock** report on 64
 - table locks versus 48
- row offset number 105
- row-level locking. *See* data-only locking

S

- scan session 47
- scanning
 - skipping uncommitted transactions 32
- scans, table
 - avoiding 91
- search conditions
 - clustered indexes and 127
 - locking 9
- select** 97
 - clustered indexes and 97
 - nonclustered indexes and 107
 - optimizing 119
 - queries 35
 - skipping uncommitted transactions 32
- serial query processing
 - demand locks and 13
- serializable reads
 - phantoms and 17
- set** command
 - transaction isolation level** 82
- shared** keyword
 - cursors and 88
 - locking and 88
- shared locks
 - cursors and 88
 - holdlock** keyword 85
 - page 9
 - sp_lock** report on 64
 - table 10
- size
 - nonclustered and clustered indexes 105
- skip
 - nonqualifying rows 33
- sleeping locks 61
- sort operations (**order by**)
 - indexing to avoid 91
- sp_chgattribute**, added option for optimistic index locking 57
- sp_droplockpromote** 52
- sp_droprowlockpromote** 52
- sp_help**, displays optimistic index locking 57
- sp_lock** 63
- sp_object_stats** 73–74
- sp_setpglockpromote** 51
- sp_setrowlockpromote** 51
- sp_who**
 - blocking process 61
- space
 - clustered compared to nonclustered indexes 105
- space allocation
 - clustered index creation 125
 - deallocation of index pages 105
 - index page splits 101
 - monotonically increasing key values and 101
 - page splits and 99
- splitting
 - data pages on inserts 99
- SQL standards
 - concurrency problems 44
- square brackets []
 - in SQL statements xiii
- storage management
 - space deallocation and 104
- symbols
 - in SQL statements xiii
- syntax conventions, Transact-SQL xiii

T

- table locks 7
 - controlling 19
 - page locks versus 48
 - row locks versus 48
 - sp_lock** report on 64
 - types of 10
- table scans
 - avoiding 91
 - locks and 30
- tables
 - locks held on 19, 64
 - secondary 141
- tasks
 - demand locks and 13
- testing
 - hot spots 128
 - nonclustered indexes 133
- time interval
 - deadlock checking 72
- transaction isolation level** option, **set** 82
- transaction isolation levels
 - lock duration and 26
 - or** processing and 32
- transactions
 - close on endtran** option 88
 - deadlock resolution 68
 - default isolation level 82
 - locking 3
- tsequal** system function
 - compared to **holdlock** 43

U

- uncommitted
 - inserts during selects 32
 - updates, qualifying old and new 36
- unique constraints
 - index created by 125
- unique indexes 91
 - optimizing 132
- update** command
 - transaction isolation levels and 23
- update locks 9
 - sp_lock** report on 64

- update operations
 - hot spots 42
 - index updates and 133
- user log cache, in ALS 144
- using asynchronous log service (Als) 143

V

- variable-length columns
 - index overhead and 142

W

- wait times 74
- when to use ALS 145
- where** clause
 - creating indexes for 128
- worker processes
 - deadlock detection and 69
 - locking and 14

