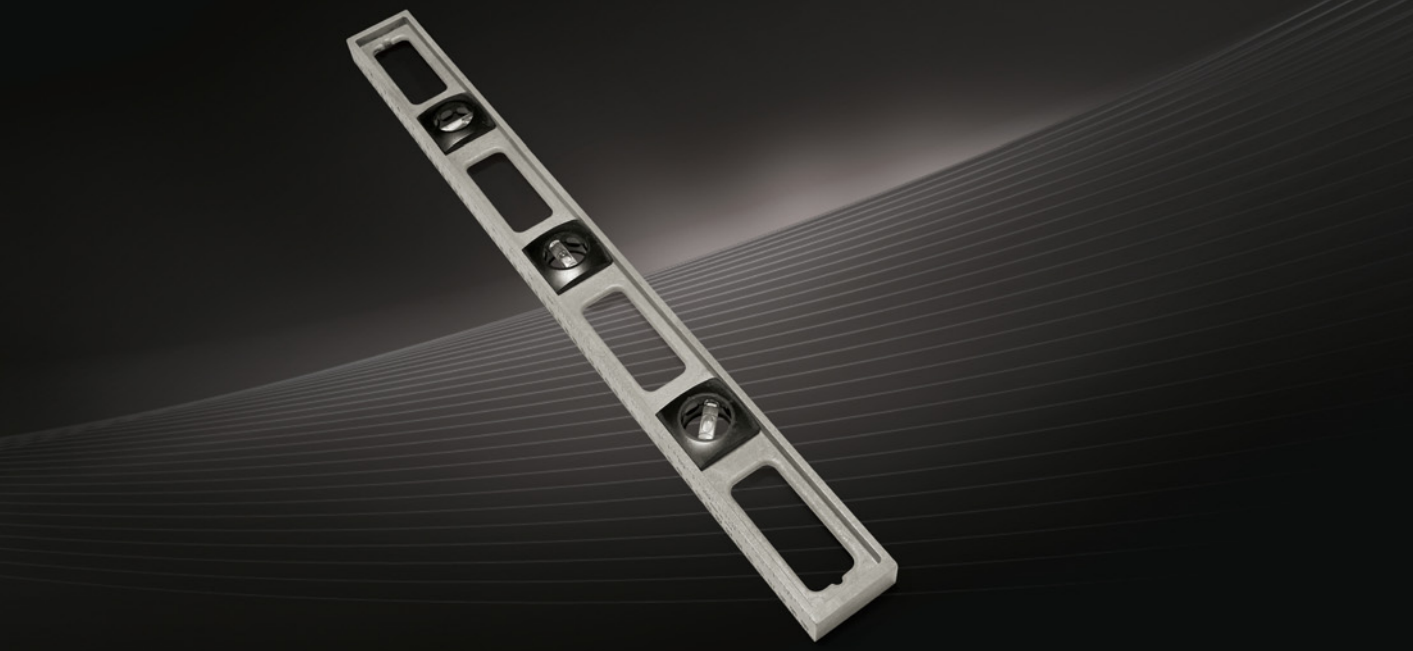
Foreword by David Campbell
Microsoft Technical Fellow

Microsoft

Microsoft® SQL Server® 2008 Internals



Kalen Delaney

Paul S. Randal, Kimberly L. Tripp,
Conor Cunningham, Adam Machanic, and Ben Nevarez

SQLTuners.net
THE DATABASE EXPERTS

Microsoft®

Microsoft® SQL Server® 2008 Internals

Kalen Delaney
Paul S. Randal, Kimberly L. Tripp
Conor Cunningham, Adam Machanic
and Ben Nevarez

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2008940524

Printed and bound in the United States of America.

2 3 4 5 6 7 8 9 10 POD 7 6 5 4 3 2

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to mspinput@microsoft.com.

Acquisitions Editor: Ken Jones
Developmental Editor: Sally Stickney
Project Editor: Lynn Finnel
Editorial Production: S4Carlisle Publishing Services
Technical Reviewer: Benjamin Nevarez; Technical Review services provided by Content Master, a member of CM Group, Ltd.
Cover: Tom Draper Design

Body Part No. X15-32079 [2012-03-30]

For Dan, forever . . . .

—Kalen

Contents at a Glance

Table of Contents

Foreword

The developers who create products such as Microsoft SQL Server typically become experts in one area of the technology, such as access methods or query execution. They live and experience the product inside out and often know their component so deeply they acquire a "curse of knowledge": they possess so much detail about their particular domain, they find it difficult to describe their work in a way that helps customers get the most out of the product.

Technical writers who create product-focused books, on the other hand, experience a product outside in. Most of these authors acquire a broad, but somewhat shallow, surface knowledge of the products they write about and produce valuable books, usually filled with many screenshots, which help new and intermediate users quickly learn how to get things done with the product.

When the curse of knowledge meets surface knowledge, it leaves a gap where many of the great capabilities created by product developers don't get communicated in a way that allows customers, particularly intermediate to advanced users, to use a product to its full potential. This is where *Microsoft SQL Server 2008 Internals* comes in. This book, like those in the earlier "Inside SQL Server" series, is the definitive reference for how SQL Server really works. Kalen Delaney has been working with the SQL Server product team for over a decade, spending countless hours with developers breaking through the curse of knowledge and then capturing the result in an incredibly clear form that allows intermediate to advanced users to wring the most from the capabilities of SQL Server. In *Microsoft SQL Server 2008 Internals,* Kalen is joined by four SQL Server experts who also share the gift of breaking the curse. Conor Cunningham and Paul Randal have years of experience as SQL Server product developers, and each of them is both a deep technical expert and a gifted communicator. Kimberly Tripp and Adam Machanic both combine a passion to really understand how things work and to then effectively share it with others. Kimberly and Adam are both standing-room-only speakers at SQL Server events. This team has captured and incorporated the details of key architectural changes for SQL Server 2008, resulting in a new, comprehensive internals reference for SQL Server.

There is a litmus test you can use to determine if a technical product title deserves a "definitive reference" classification. It's a relatively easy test but a hard one for everybody to conduct. The test, quite simply, is to look at how many of the developers who created the product in question have a copy of the book on their shelves—and reference it. I can assure you that each version of *Inside Microsoft SQL Server* that Kalen has produced has met this test. *Microsoft SQL Server 2008 Internals* will, too.

Dave Campbell

Technical Fellow
Microsoft SQL Server

Introduction

The book you are now holding is the evolutionary successor to the *Inside SQL Server* series, which included *Inside SQL Server 6.5, Inside SQL Server 7, Inside SQL Server 2000,* and *Inside SQL Server 2005* (in four volumes). The *Inside* series was becoming too unfocused, and the name "Inside" had been usurped by other authors and even other publishers. I needed a title that was much more indicative of what this book is really about.

SQL Server 2008 Internals tells you how SQL Server, Microsoft's flagship relational database product, works. Along with that, I explain how you can use the knowledge of how it works to help you get better performance from the product, but that is a side effect, not the goal. There are dozens of other books on the market that describe tuning and best practices for SQL Server. This one helps you understand why certain tuning practices work the way they do, and it helps you determine your own best practices as you continue to work with SQL Server as a developer, data architect, or DBA.

Who This Book Is For

This book is intended to be read by anyone who wants a deeper understanding of what SQL Server does behind the scenes. The focus of this book is on the core SQL Server engine—in particular, the query processor and the storage engine. I expect that you have some experience with both the SQL Server engine and with the T-SQL language. You don't have to be an expert in either, but it helps if you aspire to become an expert and would like to find out all you can about what SQL Server is actually doing when you submit a query for execution.

This series doesn't discuss client programming interfaces, heterogeneous queries, business intelligence, or replication. In fact, most of the high-availability features are not covered, but a few, such as mirroring, are mentioned at a high level when we discuss database property settings. I don't drill into the details of some internal operations, such as security, because that's such a big topic it deserves a whole volume of its own.

My hope is that you'll look at the cup as half full instead of half empty and appreciate this book for what it does include. As for the topics that aren't included, I hope you'll find the information you need in other sources.

What This Book Is About

SQL Server Internals provides detailed information on the way that SQL Server processes your queries and manages your data. It starts with an overview of the architecture of the SQL Server relational database system and then continues looking at aspects of query processing and data storage in 10 additional chapters, as follows:

- Chapter 1 SQL Server 2008 Architecture and Configuration
- Chapter 2 Change Tracking, Tracing, and Extended Events
- Chapter 3 Databases and Database Files
- Chapter 4 Logging and Recovery
- Chapter 5 Tables
- Chapter 6 Indexes: Internals and Management
- Chapter 7 Special Storage
- Chapter 8 The Query Optimizer
- Chapter 9 Plan Caching and Recompilation
- Chapter 10 Transactions and Concurrency
- Chapter 11 DBCC Internals

A twelfth chapter covering the details of reading query plans is available in the companion content (which is described in the next section). This chapter, called "Query Execution," was part of my previous book, *Inside SQL Server 2005: Query Tuning and Optimization*. Because 99 percent of the chapter is still valid for SQL Server 2008, we have included it "as is" for your additional reference.

Companion Content

This book features a companion Web site that makes available to you all the code used in the book, organized by chapter. The companion content also includes an extra chapter from my previous book, as well as the "History of SQL Server" chapter from my book *SQL Server 2000*. The site also provides extra scripts and tools to enhance your experience and understanding of SQL Server internals. As errors are found and reported, they will also be posted online. You can access this content from the companion site at this address: *http://www.SQLServerInternals.com/companion*.

System Requirements

To use the code samples, you'll need Internet access and a system capable of running SQL Server 2008 Enterprise or Developer edition. To get system requirements for SQL Server 2008 and to obtain a trial version, go to *http://www.microsoft.com/downloads*.

Support for This Book

Every effort has been made to ensure the accuracy of this book and the contents of the companion Web site. As corrections or changes are collected, they will be added to a Microsoft Knowledge Base article.

Microsoft Press provides support for books at the following Web site:

http://www.microsoft.com/learning/support/books/

Questions and Comments

If you have comments, questions, or ideas regarding the book, or questions that are not answered by visiting the sites above, please send them to Microsoft Press via e-mail to

mspinput@microsoft.com

Or via postal mail to

Microsoft Press
Attn: *Microsoft SQL Server 2008 Internals* Editor
One Microsoft Way
Redmond, WA 98052-6399

Please note that Microsoft software product support is not offered through the above addresses.

Acknowledgments

As always, a work like this is not an individual effort, and for this current volume, it is truer than ever. I was honored to have four other SQL Server experts join me in writing *SQL Server 2008 Internals*, and I truly could not have written this book alone. I am grateful to Adam Machanic, Paul Randal, Conor Cunningham, and Kimberly Tripp for helping to make this book a reality. In addition to my brilliant co-authors, this book could never have seen the light of day with help and encouragement from many other people.

First on my list is you, the readers. Thank you to all of you for reading what I have written. Thank you to those who have taken the time to write to me about what you thought of the book and what else you want to learn about SQL Server. I wish I could answer every question in detail. I appreciate all your input, even when I'm unable to send you a complete reply. One particular reader of one of my previous books, *Inside SQL Server 2005: The Storage Engine*, deserves particular thanks. I came to know Ben Nevarez as a very astute reader who found some uncaught errors and subtle inconsistencies and politely and succinctly reported them to me through my Web site. After a few dozen e-mails, I started to look forward to Ben's e-mails and was delighted when I finally got the chance to meet him. Ben is now my most valued technical reviewer, and I am deeply indebted to him for his extremely careful reading of every one of the chapters.

As usual, the SQL Server team at Microsoft has been awesome. Although Lubor Kollar and Sunil Agarwal were not directly involved in much of the research for this book, I always knew they were there in spirit, and both of them always had an encouraging word whenever I saw them.

Boris Baryshnikov, Kevin Farlee, Marcel van der Holst, Peter Byrne, Sangeetha Shekar, Robin Dhamankar, Artem Oks, Srini Acharya, and Ryan Stonecipher met with me and responded to my (sometimes seemingly endless) e-mails. Jerome Halmans, Joanna Omel, Nikunj Koolar, Tres London, Mike Purtell, Lin Chan, and Dipti Sangani also offered valuable technical insights and information when responding to my e-mails. I hope they all know how much I appreciated every piece of information I received.

I am also indebted to Bob Ward, Bob Dorr, and Keith Elmore of the SQL Server Product Support team, not just for answering occasional questions but for making so much information about SQL Server available through white papers, conference presentations, and Knowledge Base articles. I am grateful to Alan Brewer and Gail Erickson for the great job they and their User Education team did putting together the SQL Server documentation in *SQL Server Books Online*.

And, of course, Buck Woody deserves my gratitude many times over. First from his job in the User Education group, then as a member of the SQL Server development team, he was always there when I had an unanswered question. His presentations and blog posts are always educational as well as entertaining, and his generosity and unflagging good spirits are a true inspiration.

I would also like to thank Leona Lowry and Cheryl Walter for finding me office space in the same building as most of the SQL Server team. The welcome they gave me was much appreciated.

I would like to extend my heartfelt thanks to all of the SQL Server MVPs, but most especially Erland Sommarskog. Erland wrote the section in Chapter 5 on collations just because he thought it was needed, and that someone who has to deal with only the 26 letters of the English alphabet could never do it justice. Also deserving of special mention are Tibor Karaszi and Roy Harvey, for all the personal support and encouragement they gave me. Other MVPs who inspired me during the writing of this volume are Tony Rogerson, John Paul Cook, Steve Kass, Paul Nielsen, Hugo Kornelis, Tom Moreau, and Linchi Shea. Being a part of the SQL Server MVP team continues to be one of the greatest honors and privileges of my professional life.

I am deeply indebted to my students in my "SQL Server Internals" classes, not only for their enthusiasm for the SQL Server product and for what I have to teach and share with them, but for all they have to share with me. Much of what I have learned has been inspired by questions from my curious students. Some of my students, such as Cindy Gross and Lara Rubbelke, have become friends (in addition to becoming Microsoft employees) and continue to provide ongoing inspiration.

Most important of all, my family continues to provide the rock-solid foundation I need to do the work that I do. My husband, Dan, continues to be the guiding light of my life after 24 years of marriage. My daughter, Melissa, and my three sons, Brendan, Rickey, and Connor,

are now for the most part all grown, and are all generous, loving, and compassionate people. I feel truly blessed to have them in my life.

Kalen Delaney

Paul Randal

I've been itching to write a complete description of what DBCC CHECKDB does for many years now—not least to get it all out of my head and make room for something else! When Kalen asked me to write the "Consistency Checking" chapter for this book, I jumped at the chance, and for that my sincere thanks go to Kalen. I'd like to give special thanks to two people from Microsoft, among the many great folks I worked with there (and in many cases still do). The first is Ryan Stonecipher, who I hired away from being an Escalation Engineer in SQL Product Support in late 2003 to work with me on DBCC, and who was suddenly thrust into complete ownership of 100,000+ lines of DBCC code when I become the team manager two months later. I couldn't have asked for more capable hands to take over my precious DBCC.... The second is Bob Ward, who heads up the SQL Product Support team and has been a great friend since my early days at Microsoft. We must have collaborated on hundreds of cases of corruption over the years, and I've yet to meet someone with more drive for solving customer problems and improving Microsoft SQL Server. Thanks must also go to Steve Lindell, the author of the original online consistency checking code for SQL Server 2000, who spent many hours patiently explaining how it worked in 1999. Finally, I'd like to thank my wife, Kimberly, who is, along with Katelyn and Kiera, the other passions in my life apart from SQL Server.

Kimberly Tripp

First, I want to thank my good friend Kalen, for inviting me to participate in this title. After working together in various capacities—even having formed a company together back in 1996—it's great to finally have our ideas and content together in a book as deep and technical as this. In terms of performance tuning, indexes are critical; there's no better way to improve a system than by creating the *right* indexes. However, knowing what's right takes multiple components, some of which is only known after experience, after testing, and after seeing something in action. For this, I want to thank many of you—readers, students, conference attendees, customers—those of you who have asked the questions, shown me interesting scenarios, and stayed late to "play" and/or just figure it out. It's the deep desire to know why something is working the way that it is that keeps this product interesting to me and has always made me want to dive deeper and deeper into understanding what's really going on. For that, I thank the SQL team in general—the folks that I've met and worked with over the years have been inspiring, intelligent, and insightful. Specifically, I want to thank a few folks on the SQL team who have patiently, quickly, and thoroughly responded to questions about what's really going on and often, why: Conor Cunningham,

Cesar Galindo-Legaria, and from my early days with SQL Server, Dave Campbell, Nigel Ellis, and Rande Blackman. Gert E. R. Drapers requires special mention due to the many hours spent together over the years where we talked, argued, and figured it out. And, to Paul, my best friend and husband, who before that *was* also a good source of SQL information. We just don't talk about it anymore . . . at home. OK, maybe a little.

Conor Cunningham

I'd like to thank Bob Beauchemin and Milind Joshi for their efforts to review my chapter, "The Query Optimizer," in this book for technical correctness. I'd also like to thank Kimberly Tripp and Paul Randal for their encouragement and support while I wrote this chapter. Finally, I'd like to thank all the members of the SQL Server Query Processor team who answered many technical questions for me.

Adam Machanic

I would like to, first and foremost, extend my thanks to Kalen Delaney for leading the effort of this book from conception through reality. Kalen did a great job of keeping us focused and on task, as well as helping to find those hidden nuggets of information that make a book like this one great. A few Microsoft SQL Server team members dedicated their time to helping review my work: Jerome Halmans and Fabricio Voznika from the Extended Events team, and Mark Scurrell from the Change Tracking team. I would like to thank each of you for keeping me honest, answering my questions, and improving the quality of my chapter. Finally, I would like to thank Kate and Aura, my wife and daughter, who always understand when I disappear into the office for a day or two around deadline time.

Chapter 3

Databases and Database Files

Kalen Delaney

Simply put, a Microsoft SQL Server database is a collection of objects that hold and manipulate data. A typical SQL Server instance has only a handful of databases, but it's not unusual for a single instance to contain several dozen databases. The technical limit for one SQL Server instance is 32,767 databases. But practically speaking, this limit would never be reached.

To elaborate a bit, you can think of a SQL Server database as having the following properties and features:

- It is a collection of many objects, such as tables, views, stored procedures, and constraints. The technical limit is 2^{31}–1 (more than 2 billion) objects. The number of objects typically ranges from hundreds to tens of thousands.
- It is owned by a single SQL Server login account.
- It maintains its own set of user accounts, roles, schemas, and security.
- It has its own set of system tables to hold the database catalog.
- It is the primary unit of recovery and maintains logical consistency among objects within it. (For example, primary and foreign key relationships always refer to other tables within the same database, not in other databases.)
- It has its own transaction log and manages its own transactions.
- It can span multiple disk drives and operating system files.
- It can range in size from 2 MB to a technical limit of 524,272 terabytes.
- It can grow and shrink, either automatically or manually.
- It can have objects joined in queries with objects from other databases in the same SQL Server instance or on linked servers.
- It can have specific properties enabled or disabled. (For example, you can set a database to be read-only or to be a source of published data in replication.)

And here is what a SQL Server database is *not*:

- It is not synonymous with an entire SQL Server instance.
- It is not a single SQL Server table.
- It is not a specific operating system file.

Although a database isn't the same thing as an operating system file, it always exists in two or more such files. These files are known as SQL Server *database files* and are specified either at the time the database is created, using the *CREATE DATABASE* command, or afterward, using the *ALTER DATABASE* command.

System Databases

A new SQL Server 2008 installation always includes four databases: *master*, *model*, *tempdb*, and *msdb*. It also contains a fifth, "hidden" database that you never see using any of the normal SQL commands that list all your databases. This database is referred to as the *resource database*, but its actual name is *mssqlsystemresource*.

master

The *master* database is composed of system tables that keep track of the server installation as a whole and all other databases that are subsequently created. Although every database has a set of system catalogs that maintain information about objects that the database contains, the *master* database has system catalogs that keep information about disk space, file allocations and usage, system-wide configuration settings, endpoints, login accounts, databases on the current instance, and the existence of other servers running SQL Server (for distributed operations).

The *master* database is critical to your system, so always keep a current backup copy of it. Operations such as creating another database, changing configuration values, and modifying login accounts all make modifications to *master*, so you should always back up *master* after performing such actions.

model

The *model* database is simply a template database. Every time you create a new database, SQL Server makes a copy of *model* to form the basis of the new database. If you'd like every new database to start out with certain objects or permissions, you can put them in *model*, and all new databases inherit them. You can also change most properties of the *model* database by using the *ALTER DATABASE* command, and those property values then are used by any new database you create.

tempdb

The *tempdb* database is used as a workspace. It is unique among SQL Server databases because it's re-created—not recovered—every time SQL Server is restarted. It's used for temporary tables explicitly created by users, for worktables that hold intermediate results created internally by SQL Server during query processing and sorting, for maintaining row versions used in snapshot

isolation and certain other operations, and for materializing static cursors and the keys of keyset cursors. Because the *tempdb* database is re-created, any objects or permissions that you create in the database are lost the next time you start your SQL Server instance. An alternative is to create the object in the *model* database, from which *tempdb* is copied. (Keep in mind that any objects that you create in the *model* database also are added to any new databases you create subsequently. If you want objects to exist only in *tempdb,* you can create a startup stored procedure that creates the objects every time your SQL Server instance starts.)

The *tempdb* database sizing and configuration is critical for optimal functioning and performance of SQL Server, so I'll discuss *tempdb* in more detail in its own section later in this chapter.

The Resource Database

As mentioned, the *mssqlsystemresource* database is a hidden database and is usually referred to as the *resource database*. Executable system objects, such as system stored procedures and functions, are stored here. Microsoft created this database to allow very fast and safe upgrades. If no one can get to this database, no one can change it, and you can upgrade to a new service pack that introduces new system objects by simply replacing the resource database with a new one. Keep in mind that you can't see this database using any of the normal means for viewing databases, such as selecting from *sys.databases* or executing *sp_helpdb*. It also won't show up in the system databases tree in the Object Explorer pane of SQL Server Management Studio, and it does not appear in the drop-down list of databases accessible from your query windows. However, this database still needs disk space.

You can see the files in your default *binn* directory by using Microsoft Windows Explorer. My data directory is at C:\Program Files\Microsoft SQL Server\MSSQL10.MSSQLSERVER\MSSQL\Binn; I can see a file called mssqlsystemresource.mdf, which is 60.2 MB in size, and mssqlsystemresource.ldf, which is 0.5 MB. The created and modified date for both of these files is the date that the code for the current build was frozen. It should be the same date that you see when you run *SELECT @@version.* For SQL Server 2008, Service Pack 1, this is Mar 29 2009.

If you have a burning need to "see" the contents of *mssqlsystemresource*, a couple of methods are available. The easiest, if you just want to see what's there, is to stop SQL Server, make copies of the two files for the resource database, restart SQL Server, and then attach the copied files to create a database with a new name. You can do this by using Object Explorer in Management Studio or by using the *CREATE DATABASE FOR ATTACH* syntax to create a clone database, as shown here:

```
CREATE DATABASE resource_COPY
ON (NAME = data, FILENAME = 'C:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\binn
        \mssqlsystemresource_COPY.mdf'),
   (NAME = log, FILENAME =
    'C:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\binn\mssqlsystemresource_COPY.ldf')
    FOR ATTACH;
```

SQL Server treats this new *resource_COPY* database like any other user database, and it does not treat the objects in it as special in any way. If you want to change anything in the resource database, such as the text of a supplied system stored procedure, changing it in *resource_COPY* obviously does not affect anything else on your instance. However, if you start your SQL Server instance in single-user mode, you can make a single connection to your SQL Server, and that connection can use the *mssqlsystemresource* database. Starting an instance in single-user mode is not the same thing as setting a database to single-user mode. For details on how to start SQL Server in single-user mode, see the *SQL Server Books Online* entry for the sqlservr.exe application. In Chapter 6, "Indexes: Internals and Management," when I discuss database objects, I'll discuss some of the objects in the resource database.

msdb

The *msdb* database is used by the SQL Server Agent service and other companion services, which perform scheduled activities such as backups and replication tasks, and the Service Broker, which provides queuing and reliable messaging for SQL Server. In addition to backups, objects in *msdb* support jobs, alerts, log shipping, policies, database mail, and recovery of damaged pages. When you are not actively performing these activities on this database, you can generally ignore *msdb*. (But you might take a peek at the backup history and other information kept there.) All the information in *msdb* is accessible from Object Explorer in Management Studio, so you usually don't need to access the tables in this database directly. You can think of the *msdb* tables as another form of system table: Just as you can never directly modify system tables, you shouldn't directly add data to or delete data from tables in *msdb* unless you really know what you're doing or are instructed to do so by a SQL Server technical support engineer. Prior to SQL Server 2005, it was actually possible to drop the *msdb* database; your SQL Server instance was still usable, but you couldn't maintain any backup history, and you weren't able to define tasks, alerts, or jobs or set up replication. There is an undocumented traceflag that allows you to drop the *msdb* database, but because the default *msdb* database is so small, I recommend leaving it alone even if you think you might never need it.

Sample Databases

Prior to SQL Server 2005, the installation program automatically installed sample databases so you would have some actual data for exploring SQL Server functionality. As part of Microsoft's efforts to tighten security, SQL Server 2008 does not automatically install any sample databases. However, several sample databases are widely available.

AdventureWorks

AdventureWorks actually comprises a family of sample databases that was created by the Microsoft User Education group as an example of what a "real" database might look like. The family includes: *AdventureWorks2008, AdventureWorksDW2008,* and *AdventureWorksLT2008,*

as well as their counterparts created for SQL Server 2005: *AdventureWorks, AdventureWorksDW,* and *AdventureWorksLT*. You can download these databases from the Microsoft codeplex site at *http://www.codeplex.com/SqlServerSamples*. The database was designed to showcase SQL Server features, including the organization of objects into different schemas. These databases are based on data needed by the fictitious Adventure Works Cycles company. The *AdventureWorks* and *AdventureWorks2008* databases are designed to support OLTP applications and *AdventureWorksDW* and *AdventureWorksDW2008* are designed to support the business intelligence features of SQL Server and are based on a completely different database architecture. Both designs are highly normalized. Although normalized data and many separate schemas might map closely to a real production database's design, they can make it quite difficult to write and test simple queries and to learn basic SQL.

Database design is not a major focus of this book, so most of my examples use simple tables that I create; if more than a few rows of data are needed, I'll sometimes copy data from one or more *AdventureWorks2008* tables into tables of my own. It's a good idea to become familiar with the design of the *AdventureWorks* family of databases because many of the examples in *SQL Server Books Online* and in white papers published on the Microsoft Web site (*http://www.microsoft.com/sqlserver/2008/en/us/white-papers.aspx*) use data from these databases.

Note that it is also possible to install an *AdventureWorksLT2008* (or *AdventureWorksLT*) database, which is a highly simplified and somewhat denormalized version of the *AdventureWorks* OLTP database and focuses on a simple sales scenario with a single schema.

pubs

The *pubs* database is a sample database that was used extensively in earlier versions of SQL Server. Many older publications with SQL Server examples assume that you have this database because it was installed automatically on versions of SQL Server prior to SQL Server 2005. You can download a script for building this database from Microsoft's Web site, and I have also included the script with this book's companion content at *http://www.SQLServerInternals.com/companion*.

The *pubs* database is admittedly simple, but that's a feature, not a drawback. It provides good examples without a lot of peripheral issues to obscure the central points. You shouldn't worry about making modifications in the *pubs* database as you experiment with SQL Server features. You can rebuild the *pubs* database from scratch by running the supplied script. In a query window, open the file named Instpubs.sql and execute it. Make sure there are no current connections to *pubs* because the current *pubs* database is dropped before the new one is created.

Northwind

The *Northwind* database is a sample database that was originally developed for use with Microsoft Office Access. Much of the pre–SQL Server 2005 documentation dealing with application programming uses *Northwind*. *Northwind* is a bit more complex than *pubs*, and, at almost 4 MB, it is slightly larger. As with *pubs*, you can download a script from the

Microsoft Web site to build it, or you can use the script provided with the companion content. The file is called Instnwnd.sql. In addition, some of the sample scripts for this book use a modified copy of *Northwind* called *Northwind2*.

Database Files

A database file is nothing more than an operating system file. (In addition to database files, SQL Server also has *backup devices,* which are logical devices that map to operating system files or to physical devices such as tape drives. In this chapter, I won't be discussing files that are used to store backups.) A database spans at least two, and possibly several, database files, and these files are specified when a database is created or altered. Every database must span at least two files, one for the data (as well as indexes and allocation pages) and one for the transaction log.

SQL Server 2008 allows the following three types of database files:

- **Primary data files** Every database has one primary data file that keeps track of all the rest of the files in the database, in addition to storing data. By convention, a primary data file has the extension .mdf.
- **Secondary data files** A database can have zero or more secondary data files. By convention, a secondary data file has the extension .ndf.
- **Log files** Every database has at least one log file that contains the information necessary to recover all transactions in a database. By convention, a log file has the extension .ldf.

In addition, SQL Server 2008 databases can have filestream data files and full-text data files. Filestream data files will be discussed in the section "Filestream Filegroups," later in this chapter, and in Chapter 7, "Special Storage." Full-text data files are created and managed completely, separately from your other database files and are beyond the scope of this book.

Each database file has five properties that can be specified when you create the file: a logical filename, a physical filename, an initial size, a maximum size, and a growth increment. (Filestream data files have only the logical and physical name properties.) The value of these properties, along with other information about each file, can be seen through the metadata view *sys.database_files*, which contains one row for each file used by a database. Most of the columns shown in *sys.database_files* are listed in Table 3-1. The columns not mentioned here contain information dealing with transaction log backups relevant to the particular file, and I'll discuss the transaction log in Chapter 4, "Logging and Recovery."

TABLE 3-1 The *sys.database_files* Catalog View

Column	Description
fileid	The file identification number (unique for each database).
file_guid	GUID for the file. NULL = Database was upgraded from an earlier version of SQL Server.

TABLE 3-1 The *sys.database_files* Catalog View

Column	Description
type	File type: 0 = Rows (includes full-text catalogs upgraded to or created in SQL Server 2008) 1 = Log 2 = FILESTREAM 3 = Reserved for future use 4 = Full-text (includes full-text catalogs from versions earlier than SQL Server 2008)
type_desc	Description of the file type: ROWS LOG FILESTREAM FULLTEXT
data_space_id	ID of the data space to which this file belongs. Data space is a filegroup. 0 = Log file.
name	The logical name of the file.
physical_name	Operating-system file name.
state	File state: 0 = ONLINE 1 = RESTORING 2 = RECOVERING 3 = RECOVERY_PENDING 4 = SUSPECT 5 = Reserved for future use 6 = OFFLINE 7 = DEFUNCT
state_desc	Description of the file state: ONLINE RESTORING RECOVERING RECOVERY_PENDING SUSPECT OFFLINE DEFUNCT
size	Current size of the file, in 8-KB pages. 0 = Not applicable For a database snapshot, size reflects the maximum space that the snapshot can ever use for the file.

TABLE 3-1 The *sys.database_files* Catalog View

Column	Description
max_size	Maximum file size, in 8-KB pages: 0 = No growth is allowed. –1 = File will grow until the disk is full. 268435456 = Log file will grow to a maximum size of 2 terabytes.
growth	0 = File is a fixed size and will not grow. >0 = File will grow automatically. If *is_percent_growth* = 0, growth increment is in units of 8-KB pages, rounded to the nearest 64 KB. If *is_percent_growth* = 1, growth increment is expressed as a whole number percentage.
is_media_read_only	1 = File is on read-only media. 0 = File is on read/write media.
is_read_only	1 = File is marked read-only. 0 = File is marked read/write.
is_sparse	1 = File is a sparse file. 0 = File is not a sparse file. (Sparse files are used with database snapshots, discussed later in this chapter.)
is_percent_growth	See description for *growth* column, above.
is_name_reserved	1 = Dropped file name (name or physical_name) is reusable only after the next log backup. When files are dropped from a database, the logical names stay in a reserved state until the next log backup. This column is relevant only under the full recovery model and the bulk-logged recovery model.

Creating a Database

The easiest way to create a database is to use Object Explorer in Management Studio, which provides a graphical front end to the T-SQL commands that actually create the database and set its properties. Figure 3-1 shows the New Database dialog box, which represents the T-SQL *CREATE DATABASE* command for creating a new user database. Only someone with the appropriate permissions can create a database, either through Object Explorer or by using the *CREATE DATABASE* command. This includes anyone in the *sysadmin* role, anyone who has been granted CONTROL or ALTER permission on the server, and any user who has been granted CREATE DATABASE permission by someone with the *sysadmin* or *dbcreator* role.

When you create a new database, SQL Server copies the *model* database. If you have an object that you want created in every subsequent user database, you should create that object in *model* first. You can also use *model* to set default database options in all subsequently created

databases. The *model* database includes 53 objects—45 system tables, 6 objects used for SQL Server Query Notifications and Service Broker, 1 table used for helping to manage filestream data, and 1 table for helping to manage change tracking. You can see these objects by selecting from the system table called *sys.objects*. However, if you run the procedure *sp_help* in the *model* database, it will list 1,978 objects. It turns out that most of these objects are not really stored in the *model* database but are accessible through it. In Chapter 5, "Tables," I'll tell you what the other kinds of objects are and how you can tell whether an object is really stored in a particular database. Most of the objects you see in *model* will show up when you run *sp_help* in any database, but your user databases will probably have more objects added to this list. The contents of *model* are just the starting point.

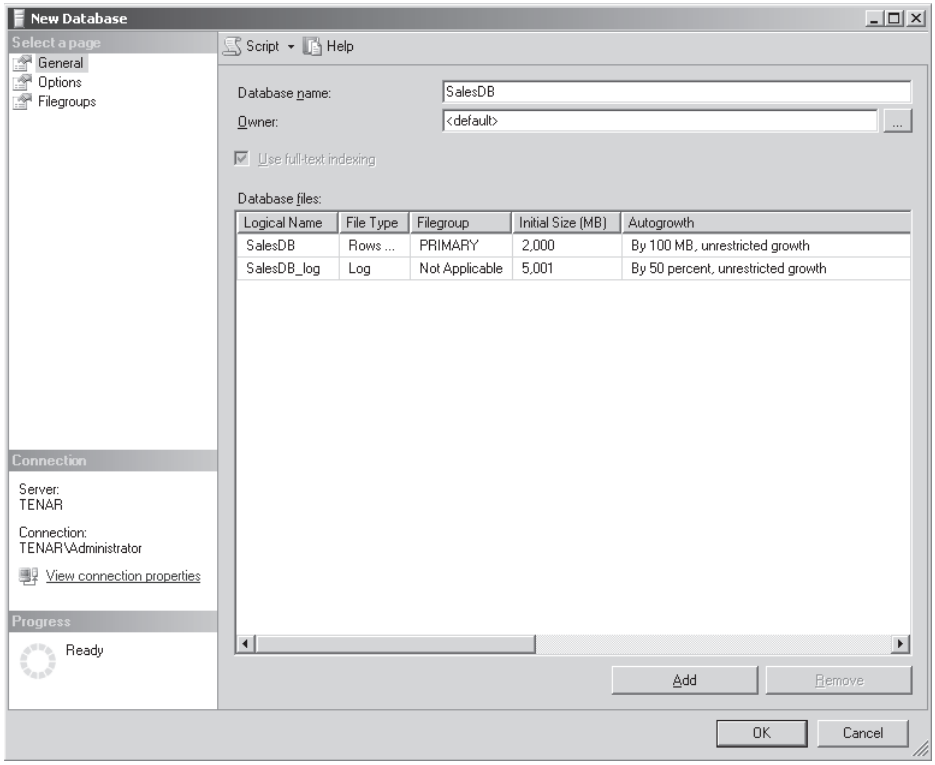


FIGURE 3-1 The New Database dialog box, where you can create a new database

A new user database must be 3 MB or larger (including the transaction log), and the primary data file size must be at least as large as the primary data file of the *model* database. (The *model* database only has one file and cannot be altered to add more. So the size of the primary data file and the size of the database are basically the same for *model*.) Almost all the possible arguments to the *CREATE DATABASE* command have default values, so it's possible to create a database using a simple form of *CREATE DATABASE*, such as this:

```
CREATE DATABASE newdb;
```

This command creates the *newdb* database, with a default size, on two files whose logical names—*newdb* and *newdb_log*—are derived from the name of the database. The corresponding physical files, newdb.mdf and newdb_log.ldf, are created in the default data directory, which is usually determined at the time SQL Server is installed.

The SQL Server login account that created the database is known as the *database owner*, and that information is stored with the information about the database properties in the *master* database. A database can have only one actual owner, who always corresponds to a login name. Any login that uses any database has a user name in that database, which might be the same name as the login name but doesn't have to be. The login that is the owner of a database always has the special user name *dbo* when using the database it owns. I'll discuss database users later in this chapter when I tell you about the basics of database security. The default size of the data file is the size of the primary data file of the *model* database (which is 2 MB by default), and the default size of the log file is 0.5 MB. Whether the database name, *newdb*, is case-sensitive depends on the sort order that you chose during setup. If you accepted the default, the name is case-insensitive. (Note that the actual command *CREATE DATABASE* is case-insensitive, regardless of the case sensitivity chosen for data.)

Other default property values apply to the new database and its files. For example, if the LOG ON clause is not specified but data files are specified, SQL Server creates a log file with a size that is 25 percent of the sum of the sizes of all data files.

If the MAXSIZE clause isn't specified for the files, the file grows until the disk is full. (In other words, the file size is considered unlimited.) You can specify the values for *SIZE*, *MAXSIZE*, and *FILEGROWTH* in units of terabytes, GB, and MB (the default), or KB. You can also specify the *FILEGROWTH* property as a percentage. A value of 0 for *FILEGROWTH* indicates no growth. If no *FILEGROWTH* value is specified, the default growth increment for data files is 1 MB. The log file *FILEGROWTH* default is specified as 10 percent.

A *CREATE DATABASE* Example

The following is a complete example of the *CREATE DATABASE* command, specifying three files and all the properties of each file:

```
CREATE DATABASE Archive
ON
PRIMARY
( NAME = Arch1,
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\archdat1.mdf',
SIZE = 100MB,
MAXSIZE = 200MB,
FILEGROWTH = 20MB),
( NAME = Arch2,
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\archdat2.ndf',
```

```
SIZE = 10GB,
MAXSIZE = 50GB,
FILEGROWTH = 250MB)
LOG ON
( NAME = Archlog1,
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\archlog1.ldf',
SIZE = 2GB,
MAXSIZE = 10GB,
FILEGROWTH = 100MB);
```

Expanding or Shrinking a Database

Databases can be expanded and shrunk automatically or manually. The mechanism for automatic expansion is completely different from the mechanism for automatic shrinkage. Manual expansion is also handled differently from manual shrinkage. Log files have their own rules for growing and shrinking; I'll discuss changes in log file size in Chapter 4.

Warning Shrinking a database or any data file is an extremely resource-intensive operation, and the only reason to do it is if you absolutely must reclaim disk space. Shrinking a data file can also lead to excessive logical fragmentation within your database. We'll discuss fragmentation in Chapter 6 and shrinking in Chapter 11, "DBCC Internals."

Automatic File Expansion

Expansion can happen automatically to any one of the database's files when that particular file becomes full. The file property *FILEGROWTH* determines how that automatic expansion happens. The *FILEGROWTH* property that is specified when the file is first defined can be qualified using the suffix *TB, GB, MB*, *KB*, or *%*, and it is always rounded up to the nearest 64 KB. If the value is specified as a percentage, the growth increment is the specified percentage of the size of the file when the expansion occurs. The file property *MAXSIZE* sets an upper limit on the size.

Allowing SQL Server to grow your data files automatically is no substitute for good capacity planning before you build or populate any tables. Enabling autogrow might prevent some failures due to unexpected increases in data volume, but it can also cause problems. If a data file is full and your autogrow percentage is set to grow by 10 percent, if an application attempts to insert a single row and there is no space, the database might start to grow by a large amount (10 percent of 10,000 MB is 1,000 MB). This in itself can take a lot of time if fast file initialization (discussed in the next section) is not being used. The growth might take so long that the client application's timeout value is exceeded, which means the insert query fails. The query would have failed anyway if autogrow weren't set, but with autogrow enabled, SQL Server spends a lot of time trying to grow the file, and you won't be informed of the problem immediately. In addition, file growth can result in physical fragmentation on the disk.

With autogrow enabled, your database files still cannot grow the database size beyond the limits of the available disk space on the drives on which files are defined, or beyond the size specified in the *MAXSIZE* file property. So if you rely on the autogrow functionality to size your databases, you must still independently check your available hard disk space or the total file size. (The undocumented extended procedure *xp_fixeddrives* returns a list of the amount of free disk space on each of your local volumes.) To reduce the possibility of running out of space, you can watch the Performance Monitor counter SQL Server: Databases Object: Data File Size and set up a performance alert to fire when the database file reaches a certain size.

Manual File Expansion

You can expand a database file manually by using the *ALTER DATABASE* command with the MODIFY FILE option to change the *SIZE* property of one or more of the files. When you alter a database, the new size of a file must be larger than the current size. To decrease the size of a file, you use the *DBCC SHRINKFILE* command, which I'll tell you about shortly.

Fast File Initialization

SQL Server 2008 data files (but not log files) can be initialized instantaneously. This allows for fast execution of the file creation and growth. Instant file initialization adds space to the data file without filling the newly added space with zeros. Instead, the actual disk content is overwritten only as new data is written to the files. Until the data is overwritten, there is always the chance that a hacker using an external file reader tool can see the data that was previously on the disk. Although the SQL Server 2008 documentation describes the instant file initialization feature as an "option," it is not really an option within SQL Server. It is actually controlled through a Windows security setting called SE_MANAGE_VOLUME_NAME, which is granted to Windows administrators by default. (This right can be granted to other Windows users by adding them to the Perform Volume Maintenance Tasks security policy.) If your SQL Server service account is in the Windows Administrator role and your SQL Server is running on a Windows XP, Windows Server 2003, or Windows Server 2008 filesystem, instant file initialization is used. If you want to make sure your database files are zeroed out as they are created and expanded, you can use traceflag 1806 or deny SE_MANAGE_VOLUME_NAME rights to the account under which your SQL Server service is running.

Automatic Shrinkage

The database property *autoshrink* allows a database to shrink automatically. The effect is the same as doing a *DBCC SHRINKDATABASE (dbname, 25)*. This option leaves 25 percent free space in a database after the shrink, and any free space beyond that is returned to the operating system. The thread that performs autoshrink shrinks databases at very frequent intervals, in some cases as often as every 30 minutes. Shrinking data files is so resource-intensive that it should be done only when there is no other way to reclaim needed disk space.

Important Automatic shrinking is never recommended. In fact, Microsoft has announced that the autoshrink option will be removed in a future version of SQL Server and you should avoid using it.

Manual Shrinkage

You can shrink a database manually using one of the following DBCC commands:

```
DBCC SHRINKFILE ( {file_name | file_id }
[, target_size][, {EMPTYFILE | NOTRUNCATE | TRUNCATEONLY} ]  )

DBCC SHRINKDATABASE (database_name [, target_percent]
[, {NOTRUNCATE | TRUNCATEONLY} ]  )
```

DBCC SHRINKFILE

DBCC SHRINKFILE allows you to shrink files in the current database. When you specify *target_size*, *DBCC SHRINKFILE* attempts to shrink the specified file to the specified size in megabytes. Used pages in the part of the file to be freed are relocated to available free space in the part of the file that is retained. For example, for a 15-MB data file, a *DBCC SHRINKFILE* with a *target_size* of 12 causes all used pages in the last 3 MB of the file to be reallocated into any free slots in the first 12 MB of the file. *DBCC SHRINKFILE* doesn't shrink a file past the size needed to store the data. For example, if 70 percent of the pages in a 10-MB data file are used, a *DBCC SHRINKFILE* command with a *target_size* of 5 shrinks the file to only 7 MB, not 5 MB.

DBCC SHRINKDATABASE

DBCC SHRINKDATABASE shrinks all files in a database but does not allow any file to be shrunk smaller than its minimum size. The minimum size of a database file is the initial size of the file (specified when the database was created) or the size to which the file has been explicitly extended or reduced, using either the *ALTER DATABASE* or *DBCC SHRINKFILE* command. If you need to shrink a database smaller than its minimum size, you should use the *DBCC SHRINKFILE* command to shrink individual database files to a specific size. The size to which a file is shrunk becomes the new minimum size.

The numeric *target_percent* argument passed to the *DBCC SHRINKDATABASE* command is a percentage of free space to leave in each file of the database. For example, if you've used 60 MB of a 100-MB database file, you can specify a shrink percentage of 25 percent. SQL Server then shrinks the file to a size of 80 MB, and you have 20 MB of free space in addition to the original 60 MB of data. In other words, the 80-MB file has 25 percent of its space free. If, on the other hand, you've used 80 MB or more of a 100-MB database file, there is no way that SQL Server can shrink this file to leave 25 percent free space. In that case, the file size remains unchanged.

Because *DBCC SHRINKDATABASE* shrinks the database on a file-by-file basis, the mechanism used to perform the actual shrinking of data files is the same as that used with *DBCC SHRINKFILE* (when a data file is specified). SQL Server first moves pages to the front of files to free up space at the end, and then it releases the appropriate number of freed pages to the operating system. The actual internal details of how data files are shrunk will be discussed in Chapter 11.

Note Shrinking a log file is very different from shrinking a data file, and understanding how much you can shrink a log file and what exactly happens when you shrink it requires an understanding of how the log is used. For this reason, I will postpone the discussion of shrinking log files until Chapter 4.

As the warning at the beginning of this section indicated, shrinking a database or any data files is a resource-intensive operation. If you absolutely need to recover disk space from the database, you should plan the shrink operation carefully and perform it when it has the least impact on the rest of the system. You should never enable the AUTOSHRINK option, which will shrink *all* the data files at regular intervals and wreak havoc with system performance. Because shrinking data files can move data all around a file, it can also introduce fragmentation, which you then might want to remove. Defragmenting your data files can then have its own impact on productivity because it uses system resources. Fragmentation and defragmentation will be discussed in Chapter 6.

It is possible for shrink operations to be blocked by a transaction that has been enabled for either of the snapshot-based isolation levels. When this happens, *DBCC SHRINKFILE* and *DBCC SHRINKDATABASE* print out an informational message to the error log every five minutes in the first hour and then every hour after that. SQL Server also provides progress reporting for the *SHRINK* commands, available through the *sys.dm_exec_requests* view. Progress reporting will be discussed in Chapter 11.

Using Database Filegroups

You can group data files for a database into filegroups for allocation and administration purposes. In some cases, you can improve performance by controlling the placement of data and indexes into specific filegroups on specific drives or volumes. The filegroup containing the primary data file is called the *primary filegroup*. There is only one primary filegroup, and if you don't ask specifically to place files in other filegroups when you create your database, *all* of your data files are in the primary filegroup.

In addition to the primary filegroup, a database can have one or more user-defined filegroups. You can create user-defined filegroups by using the *FILEGROUP* keyword in the *CREATE DATABASE* or *ALTER DATABASE* command.

Don't confuse the primary filegroup and the primary file. Here are the differences:

- The primary file is always the first file listed when you create a database, and it typically has the file extension .mdf. The one special feature of the primary file is that it has pointers into a table in the *master* database (which you can access through the catalog view *sys.database_files*) that contains information about all the files belonging to the database.
- The primary filegroup is always the filegroup that contains the primary file. This filegroup contains the primary data file and any files not put into another specific filegroup. All pages from system tables are always allocated from files in the primary filegroup.

The Default Filegroup

One filegroup always has the property of *DEFAULT*. Note that *DEFAULT* is a property of a filegroup, not a name. Only one filegroup in each database can be the default filegroup. By default, the primary filegroup is also the default filegroup. A database owner can change which filegroup is the default by using the *ALTER DATABASE* command. When creating a table or index, it is created in the default filegroup if no specific filegroup is specified.

Most SQL Server databases have a single data file in one (default) filegroup. In fact, most users probably never know enough about how SQL Server works to know what a filegroup is. As a user acquires greater database sophistication, she might decide to use multiple devices to spread out the I/O for a database. The easiest way to do this is to create a database file on a RAID device. Still, there would be no need to use filegroups. At the next level of sophistication and complexity, the user might decide that she really wants multiple files—perhaps to create a database that uses more space than is available on a single drive. In this case, she still doesn't need filegroups—she can accomplish her goals using *CREATE DATABASE* with a list of files on separate drives.

More sophisticated database administrators might decide to have different tables assigned to different drives or to use the table and index partitioning feature in SQL Server 2008. Only then will they need to use filegroups. They can then use Object Explorer in Management Studio to create the database on multiple filegroups. Then they can right-click the database name in Object Explorer and create a script of the *CREATE DATABASE* command that includes all the files in their appropriate filegroups. They can save and reuse this script when they need to re-create the database or build a similar database.

Why Use Multiple Files?

You might wonder why you would want to create a database on multiple files located on one physical drive. There's usually no performance benefit in doing so, but it gives you added flexibility in two important ways.

First, if you need to restore a database from a backup because of a disk crash, the new database must contain the same number of files as the original. For example, if your original database consisted of one large 120-GB file, you would need to restore it to

a database with one file of that size. If you don't have another 120-GB drive immediately available, you cannot restore the database. If, however, you originally created the database on several smaller files, you have added flexibility during a restoration. You might be more likely to have several 40-GB drives available than one large 120-GB drive.

Second, spreading the database onto multiple files, even on the same drive, gives you the flexibility of easily moving the database onto separate drives if you modify your hardware configuration in the future. (Please refer to the section "Moving or Copying a Database," later in this chapter, for details.)

Objects that have space allocated to them, namely tables and indexes, are created on a particular filegroup. (They can also be created on a partition scheme, which is a collection of filegroups. I'll discuss partitioning and partition schemes in Chapter 7.) If the filegroup (or partition scheme) is not specified, objects are created on the default filegroup. When you add space to objects stored in a particular filegroup, the data is stored in a *proportional fill* manner, which means that if you have one file in a filegroup with twice as much free space as another, the first file has two extents (or units of space) allocated from it for each extent allocated from the second file. (I'll discuss extents in more detail in the section entitled "Space Allocation," later in this chapter.) It's recommended that you create all of your files to be the same size to avoid the issues of proportional fill.

You can also use filegroups to allow backups of parts of the database. Because a table is created on a single filegroup, you can choose to back up just a certain set of critical tables by backing up the filegroups in which you placed those tables. You can also restore individual files or filegroups in two ways. First, you can do a partial restore of a database and restore only a subset of filegroups, which must always include the primary filegroup. The database will be online as soon as the primary filegroup has been restored, but only objects created on the restored filegroups will be available. Partial restore of just a subset of filegroups can be a solution to allow very large databases to be available within a mandated time window. Alternatively, if you have a failure of a subset of the disks on which you created your database, you can restore backups of the filegroups on those disks on top of the existing database. This method of restoring also requires that you have log backups, so I'll discuss this topic in more detail in Chapter 4.

A *FILEGROUP CREATION* Example

This example creates a database named *sal*es with three filegroups:

- The primary filegroup, with the files *salesPrimary1* and *salesPrimary2*. The *FILEGROWTH* increment for both of these files is specified as 100 MB.
- A filegroup named *SalesGroup1,* with the files *salesGrp1File1* and *salesGrp1Fi1e2*.
- A filegroup named *SalesGroup2,* with the files *salesGrp2File1* and *salesGrp2Fi1e2*.

```
CREATE DATABASE Sales
ON PRIMARY
( NAME = salesPrimary1,
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\salesPrimary1.mdf',
SIZE = 100,
MAXSIZE = 500,
FILEGROWTH = 100 ),
( NAME = salesPrimary2,
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\salesPrimary2.ndf',
SIZE = 100,
MAXSIZE = 500,
FILEGROWTH = 100 ),
FILEGROUP SalesGroup1
( NAME = salesGrp1Fi1e1,
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\salesGrp1Fi1e1.ndf',
SIZE = 500,
MAXSIZE = 3000,
FILEGROWTH = 500 ),
( NAME = salesGrp1Fi1e2,
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\salesGrp1Fi1e2.ndf',
SIZE = 500,
MAXSIZE = 3000,
FILEGROWTH = 500 ),
FILEGROUP SalesGroup2
( NAME = salesGrp2Fi1e1,
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\salesGrp2Fi1e1.ndf',
SIZE = 100,
MAXSIZE = 5000,
FILEGROWTH = 500 ),
( NAME = salesGrp2Fi1e2,
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\salesGrp2Fi1e2.ndf',
SIZE = 100,
MAXSIZE = 5000,
FILEGROWTH = 500 )
LOG ON
( NAME = 'Sales_log',
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\saleslog.ldf',
SIZE = 5MB,
MAXSIZE = 25MB,
FILEGROWTH = 5MB );
```

Filestream Filegroups

I briefly mentioned filestream storage in Chapter 1, "SQL Server 2008 Architecture and Configuration," when I talked about configuration options. Filestream filegroups can be created when you create a database, just like regular filegroups can be, but you must specify

that the filegroup is for filestream data by using the phrase CONTAINS FILESTREAM. Unlike regular filegroups, each filestream filegroup can contain only one file reference, and that file is specified as an operating system folder, not a specific file. The path up to the last folder must exist, and the last folder must not exist. So in my example, the path C:\Data must exist, but the *Reviews_FS* subfolder cannot exist when you execute the *CREATE DATABASE* command. Also unlike regular filegroups, there is no space preallocated to the filegroup and you do not specify size or growth information for the file within the filegroup. The file and filegroup will grow as data is added to tables that have been created with filestream columns:

```
CREATE DATABASE MyMovieReviews
ON
PRIMARY
  ( NAME = Reviews_data,
    FILENAME = 'c:\data\Reviews_data.mdf'),
FILEGROUP MovieReviewsFSGroup1 CONTAINS FILESTREAM
  ( NAME = Reviews_FS,
    FILENAME = 'c:\data\Reviews_FS')
LOG ON  ( NAME = Reviews_log,
    FILENAME = 'c:\data\Reviews_log.ldf');
GO
```

If you run the previous code, you should see a Filestream.hdr file and an $FSLOG folder in the C:\Data\Reviews_FS folder. The Filestream.hdr file is a FILESTREAM container header file. This file should not be modified or removed. For existing databases, you can add a filestream filegroup using *ALTER DATABASE,* which I'll cover in the next section. All data in all columns placed in the *MovieReviewsFSGroup1* is maintained and managed with individual files created in the *Reviews_FS* folder. I'll tell you more about the file organization within this folder in Chapter 7, when I talk about special storage formats.

Altering a Database

You can use the *ALTER DATABASE* command to change a database's definition in one of the following ways:

- Change the name of the database.
- Add one or more new data files to the database. If you want, you can put these files in a user-defined filegroup. All files added in a single *ALTER DATABASE* command must go in the same filegroup.
- Add one or more new log files to the database.
- Remove a file or a filegroup from the database. You can do this only if the file or filegroup is completely empty. Removing a filegroup removes all the files in it.

- Add a new filegroup to a database. (Adding files to those filegroups must be done in a separate *ALTER DATABASE* command.)
- Modify an existing file in one of the following ways:
  - Increase the value of the *SIZE* property.
  - Change the *MAXSIZE* or *FILEGROWTH* property.
  - Change the logical name of a file by specifying a *NEWNAME* property. The value of *NEWNAME* is then used as the *NAME* property for all future references to this file.
  - Change the *FILENAME* property for files, which can effectively move the files to a new location. The new name or location doesn't take effect until you restart SQL Server. For *tempdb*, SQL Server automatically creates the files with the new name in the new location; for other databases, you must move the file manually after stopping your SQL Server instance. SQL Server then finds the new file when it restarts.
- Mark the file as OFFLINE. You should set a file to OFFLINE when the physical file has become corrupted and the file backup is available to use for restoring. (There is also an option to mark the whole database as OFFLINE, which I'll discuss shortly when I talk about database properties.) Marking a file as OFFLINE allows you to indicate that you don't want SQL Server to recover that particular file when it is restarted.
- Modify an existing filegroup in one of the following ways:
  - Mark the filegroup as READONLY so that updates to objects in the filegroup aren't allowed. The primary filegroup cannot be made READONLY.
  - Mark the filegroup as READWRITE, which reverses the READONLY property.
  - Mark the filegroup as the default filegroup for the database.
  - Change the name of the filegroup.
- Change one or more database options. (I'll discuss database options later in the chapter.)

The *ALTER DATABASE* command can make only one of the changes described each time it is executed. Note that you cannot move a file from one filegroup to another.

ALTER DATABASE Examples

The following examples demonstrate some of the changes that you can make using the *ALTER DATABASE* command.

This example increases the size of a database file:

```
USE master
GO
ALTER DATABASE Test1
MODIFY FILE
( NAME = 'test1dat3',
SIZE = 2000MB);
```

The following example creates a new filegroup in a database, adds two 500-MB files to the filegroup, and makes the new filegroup the default filegroup. You need three *ALTER DATABASE* statements:

```
ALTER DATABASE Test1
ADD FILEGROUP Test1FG1;
GO
ALTER DATABASE Test1
ADD FILE
( NAME = 'test1dat4',
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\t1dat4.ndf',
SIZE = 500MB,
MAXSIZE = 1000MB,
FILEGROWTH = 50MB),
( NAME = 'test1dat5',
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\t1dat5.ndf',
SIZE = 500MB,
MAXSIZE = 1000MB,
FILEGROWTH = 50MB)
TO FILEGROUP Test1FG1;
GO
ALTER DATABASE Test1
MODIFY FILEGROUP Test1FG1 DEFAULT;
GO
```

Databases Under the Hood

A database consists of user-defined space for the permanent storage of user objects such as tables and indexes. This space is allocated in one or more operating system files.

Databases are divided into logical pages (of 8 KB each), and within each file the pages are numbered contiguously from 0 to *x,* with the value *x* being defined by the size of the file. You can refer to any page by specifying a database ID, a file ID, and a page number. When you use the *ALTER DATABASE* command to enlarge a file, the new space is added to the end of the file. That is, the first page of the newly allocated space is page *x* + 1 on the file you're enlarging. When you shrink a database by using the *DBCC SHRINKDATABASE* or *DBCC SHRINKFILE* command, pages are removed starting at the highest-numbered page in the database (at the end) and moving toward lower-numbered pages. This ensures that page numbers within a file are always contiguous.

When you create a new database using the *CREATE DATABASE* command, it is given a unique database ID, and you can see a row for the new database in the *sys.databases* view. The rows returned in *sys.databases* include basic information about each database, such as its name, *database_id,* and creation date, as well as the value for each database option that can be set with the *ALTER DATABASE* command. I'll discuss database options in more detail later in the chapter.

Space Allocation

The space in a database is used for storing tables and indexes. The space is managed in units called *extents*. An extent is made up of eight logically contiguous pages (or 64 KB of space). To make space allocation more efficient, SQL Server 2008 doesn't allocate entire extents to tables with small amounts of data. SQL Server 2008 has two types of extents:

- **Uniform extents** These are owned by a single object; all eight pages in the extent can be used only by the owning object.
- **Mixed extents** These are shared by up to eight objects.

SQL Server allocates pages for a new table or index from mixed extents. When the table or index grows to eight pages, all future allocations use uniform extents.

When a table or index needs more space, SQL Server needs to find space that's available to be allocated. If the table or index is still less than eight pages total, SQL Server must find a mixed extent with space available. If the table or index is eight pages or larger, SQL Server must find a free uniform extent.

SQL Server uses two special types of pages to record which extents have been allocated and which type of use (mixed or uniform) the extent is available for:

- **Global Allocation Map (GAM) pages** These pages record which extents have been allocated for any type of use. A GAM has a bit for each extent in the interval it covers. If the bit is 0, the corresponding extent is in use; if the bit is 1, the extent is free. After the header and other overhead are accounted for, there are 8,000 bytes, or 64,000 bits, available on the page, so each GAM can cover about 64,000 extents, or almost 4 GB of data. This means that one GAM page exists in a file for every 4 GB of file size.
- **Shared Global Allocation Map (SGAM) pages** These pages record which extents are currently used as mixed extents and have at least one unused page. Just like a GAM, each SGAM covers about 64,000 extents, or almost 4 GB of data. The SGAM has a bit for each extent in the interval it covers. If the bit is 1, the extent being used is a mixed extent and has free pages; if the bit is 0, the extent isn't being used as a mixed extent, or it's a mixed extent whose pages are all in use.

Table 3-2 shows the bit patterns that each extent has set in the GAM and SGAM pages, based on its current use.

TABLE 3-2 Bit Settings in GAM and SGAM Pages

Current Use of Extent	GAM Bit Setting	SGAM Bit Setting
Free, not in use	1	0
Uniform extent or full mixed extent	0	0
Mixed extent with free pages	0	1

There are several tools available for actually examining the bits in the GAMs and SGAMs. Chapter 5 discusses the *DBCC PAGE* command which allows you to view the contents of a SQL Server database page using a query window. Because the page numbers of the GAMs and SGAMs are known, we can just look at pages 2 or 3. If we use format 3, which gives the most details, we can see that output displays which extents are allocated and which are not. Figure 3-2 shows the last section of the output using *DBCC PAGE* with format 3 for the first GAM page of my *AdventureWorks2008* database.

```
(1:0)       - (1:24256)   =       ALLOCATED
(1:24264)   -             = NOT ALLOCATED
(1:24272)   - (1:29752)   =       ALLOCATED
(1:29760)   - (1:30168)   = NOT ALLOCATED
(1:30176)   - (1:30240)   =       ALLOCATED
(1:30248)   - (1:30256)   = NOT ALLOCATED
(1:30264)   - (1:32080)   =       ALLOCATED
(1:32088)   - (1:32304)   = NOT ALLOCATED
```

FIGURE 3-2 GAM page contents indicating allocation status of extents in a file

This output indicates that all the extents up through the one that starts on page 24,256 are allocated. This corresponds to the first 189 MB of the file. The extent starting at 24,264 is not allocated, but the next 5,480 pages are allocated.

We can also use a graphical tool called *SQL Internals Viewer* to look at which extents have been allocated. SQL Internals Viewer is a free tool available from *http://www.SQLInternalsViewer.com*, and is also available on this book's companion Web site. Figure 3-3 shows the main allocation page for my *master* database. GAMs and SGAMs have been combined in one display and indicate the status of every page, not just every extent. The green squares indicate that the SGAM is being used but the extent is not used, so there are pages available for single-page allocations. The blue blocks indicate that both the GAM bit and the SGAM bit are set, so the corresponding extent is completely unavailable. The gray blocks indicate that the extent is free.

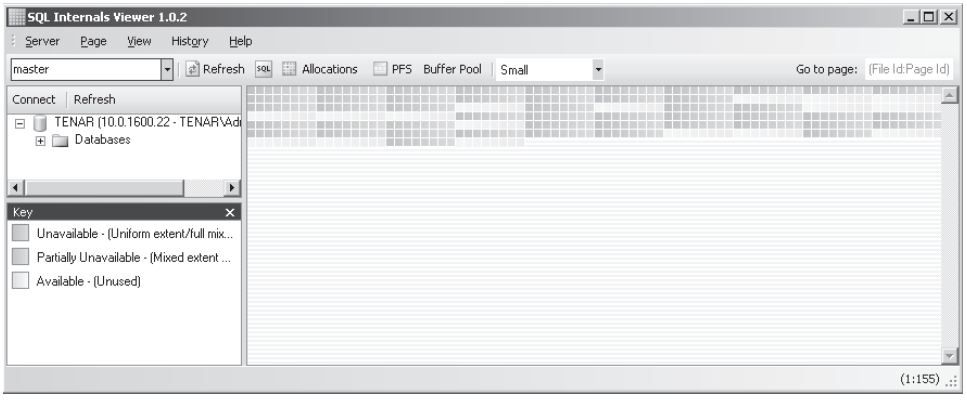


FIGURE 3-3 SQL Internals Viewer indicating the allocation status of each page

If SQL Server needs to find a new, completely unused extent, it can use any extent with a corresponding bit value of 1 in the GAM page. If it needs to find a mixed extent with available space (one or more free pages), it finds an extent with a value in the SGAM of 1 (which always has a value in the GAM of 0). If there are no mixed extents with available space, it uses the GAM page to find a whole new extent to allocate as a mixed extent, and uses one page from that. If there are no free extents at all, the file is full.

SQL Server can locate the GAMs in a file quickly because a GAM is always the third page in any database file (that is, page 2). An SGAM is the fourth page (that is, page 3). Another GAM appears every 511,230 pages after the first GAM on page 2, and another SGAM appears every 511,230 pages after the first SGAM on page 3. Page 0 in any file is the File Header page, and only one exists per file. Page 1 is a Page Free Space (PFS) page. In Chapter 5, I'll say more about how individual pages within a table look and tell you about the details of PFS pages. For now, because I'm talking about space allocation, I'll examine how to keep track of which pages belong to which tables.

IAM pages keep track of the extents in a 4-GB section of a database file used by an allocation unit. An allocation unit is a set of pages belonging to a single partition in a table or index and comprises pages of one of three storage types: pages holding regular in-row data, pages holding Large Object (LOB) data, or pages holding row-overflow data. I'll discuss these regular in-row storage in Chapter 5, and LOB, row-overflow storage, and partitions in Chapter 7.

For example, a table on four partitions that has all three types of data (in-row, LOB, and row-overflow) has at least 12 IAM pages. Again, a single IAM page covers only a 4-GB section of a single file, so if the partition spans files, there will be multiple IAM pages, and if the file is more than 4 GB in size and the partition uses pages in more than one 4-GB section, there will be additional IAM pages.

An IAM page contains a 96-byte page header, like all other pages followed by an IAM page header, which contains eight page-pointer slots. Finally, an IAM page contains a set of bits that map a range of extents onto a file, which doesn't necessarily have to be the same file that the IAM page is in. The header has the address of the first extent in the range mapped by the IAM. The eight page-pointer slots might contain pointers to pages belonging to the relevant object contained in mixed extents; only the first IAM for an object has values in these pointers. Once an object takes up more than eight pages, all of its additional extents are uniform extents—which means that an object never needs more than eight pointers to pages in mixed extents. If rows have been deleted from a table, the table can actually use fewer than eight of these pointers. Each bit of the bitmap represents an extent in the range, regardless of whether the extent is allocated to the object owning the IAM. If a bit is on, the relative extent in the range is allocated to the object owning the IAM; if a bit is off, the relative extent isn't allocated to the object owning the IAM.

For example, if the bit pattern in the first byte of the IAM is 1100 0000, the first and second extents in the range covered by the IAM are allocated to the object owning the IAM and extents 3 through 8 aren't allocated to the object owning the IAM.

IAM pages are allocated as needed for each object and are located randomly in the database file. Each IAM covers a possible range of about 512,000 pages.

The internal system view called *sys.system_internals_allocation_units* has a column called *first_iam_page* that points to the first IAM page for an allocation unit. All the IAM pages for that allocation unit are linked in a chain, with each IAM page containing a pointer to the next in the chain. You can find out more about IAMs and allocation units in Chapters 5, 6, and 7 when I discuss object and index storage.

In addition to GAMs, SGAMs, and IAMs, a database file has three other types of special allocation pages. PFS pages keep track of how each particular page in a file is used. The second page (page 1) of a file is a PFS page, as is every 8,088th page thereafter. I'll talk about them more in Chapter 5. The seventh page (page 6) is called a Differential Changed Map (DCM) page. It keeps track of which extents in a file have been modified since the last full database backup. The eighth page (page 7) is called a Bulk Changed Map (BCM) page and is used when an extent in the file is used in a minimally or bulk-logged operation. I'll tell you more about these two kinds of pages when I talk about the internals of backup and restore operations in Chapter 4. Like GAM and SGAM pages, DCM and BCM pages have 1 bit for each extent in the section of the file they represent. They occur at regular intervals—every 511,230 pages.

You can see the details of IAMs and PFS pages, as well as DCM and BCM pages, using either *DBCC PAGE* or the SQL Internals Viewer. I'll show you more examples of the output of *DBCC PAGE* in later chapters as we cover more details of the different types of allocation pages.

Setting Database Options

You can set several dozen options, or properties, for a database to control certain behavior within that database. Some options must be set to ON or OFF, some must be set to one of a list of possible values, and others are enabled by just specifying their name. By default, all the options that require ON or OFF have an initial value of OFF unless the option was set to ON in the *model* database. All databases created after an option is changed in *model* have the same values as *model*. You can easily change the value of some of these options by using Management Studio. You can set all of them directly by using the *ALTER DATABASE* command. (You can also use the *sp_dboption* system stored procedure to set some of the options, but that procedure is provided for backward compatibility only and is scheduled to be removed in the next version of SQL Server.)

Examining the *sys.databases* catalog view can show you the current values of all the options. The view also contains other useful information, such as database ID, creation date, and the Security ID (SID) of the database owner. The following query retrieves some of the most

important columns from *sys.databases* for the four databases that exist on a new default installation of SQL Server:

```
SELECT name, database_id, suser_sname(owner_sid) as owner,
     create_date, user_access_desc, state_desc
FROM sys.databases
WHERE database_id <= 4;
```

The query produces this output, although the created dates may vary:

```
name   database_id owner create_date             user_access_desc state_desc
------ ----------- ----- ----------------------- ---------------- ----------
master 1           sa    2003-04-08 09:13:36.390 MULTI_USER       ONLINE
tempdb 2           sa    2008-04-19 12:02:35.327 MULTI_USER       ONLINE
model  3           sa    2003-04-08 09:13:36.390 MULTI_USER       ONLINE
msdb   4           sa    2008-03-21 01:54:05.240 MULTI_USER       ONLINE
```

The *sys.databases* view actually contains both a number and a name for the *user_access* and *state* information. Selecting all the columns from *sys.databases* would show you that the *user_access_desc* value of MULTI_USER has a corresponding *user_access* value of 0, and the *state_desc* value of ONLINE has a *state* value of 0. *SQL Server Books Online* shows the complete list of number and name relationships for the columns in *sys.databases*. These are just two of the database options displayed in the *sys.databases* view. The complete list of database options is divided into seven main categories: state options, cursor options, auto options, parameterization options, SQL options, database recovery options, and external access options. There are also options for specific technologies that SQL Server can use, including database mirroring, Service Broker activities, change tracking, database encryption, and snapshot isolation. Some of the options, particularly the SQL options, have corresponding SET options that you can turn on or off for a particular connection. Be aware that the ODBC or OLE DB drivers turn on a number of these SET options by default, so applications act as if the corresponding database option has already been set.

Here is a list of the options, by category. Options listed on a single line and values separated by vertical bars (|) are mutually exclusive.

State options

1. SINGLE_USER | RESTRICTED_USER | MULTI_USER
2. OFFLINE | ONLINE | EMERGENCY
3. READ_ONLY | READ_WRITE

Cursor options

1. CURSOR_CLOSE_ON_COMMIT { ON | OFF }
2. CURSOR_DEFAULT { LOCAL | GLOBAL }

Auto options

1. AUTO_CLOSE { ON | OFF }
2. AUTO_CREATE_STATISTICS { ON | OFF }
3. AUTO_SHRINK { ON | OFF }
4. AUTO_UPDATE_STATISTICS { ON | OFF }
5. AUTO_UPDATE_STATISTICS_ASYNC { ON | OFF }

Parameterization options

1. DATE_CORRELATION_OPTIMIZATION { ON | OFF }
2. PARAMETERIZATION { SIMPLE | FORCED }

SQL options

1. ANSI_NULL_DEFAULT { ON | OFF }
2. ANSI_NULLS { ON | OFF }
3. ANSI_PADDING { ON | OFF }
4. ANSI_WARNINGS { ON | OFF }
5. ARITHABORT { ON | OFF }
6. CONCAT_NULL_YIELDS_NULL { ON | OFF }
7. NUMERIC_ROUNDABORT { ON | OFF }
8. QUOTED_IDENTIFIER { ON | OFF }
9. RECURSIVE_TRIGGERS { ON | OFF }

Database recovery options

1. RECOVERY { FULL | BULK_LOGGED | SIMPLE }
2. TORN_PAGE_DETECTION { ON | OFF }
3. PAGE_VERIFY { CHECKSUM | TORN_PAGE_DETECTION | NONE }

External access options

1. DB_CHAINING { ON | OFF }
2. TRUSTWORTHY { ON | OFF }

Database mirroring options

1. PARTNER { = 'partner_server' }
2. | FAILOVER

3. | FORCE_SERVICE_ALLOW_DATA_LOSS
4. | OFF
5. | RESUME
6. | SAFETY { FULL | OFF }
7. | SUSPEND
8. | TIMEOUT *integer*
9. }
10. WITNESS { = 'witness_server' }| OFF }

Service Broker options

1. ENABLE_BROKER | DISABLE_BROKER
2. NEW_BROKER
3. ERROR_BROKER_CONVERSATIONS

Change Tracking options

1. CHANGE_TRACKING {= ON [<change_tracking_settings> | = OFF}

Database Encryption options

1. ENCRYPTION {ON | OFF}

Snapshot Isolation options

1. ALLOW_SNAPSHOT_ISOLATION {ON | OFF }
2. READ_COMMITTED_SNAPSHOT {ON | OFF } [WITH <termination>]

State Options

The state options control who can use the database and for what operations. There are three aspects to usability: The user access state determines which users can use the database; the status state determines whether the database is available to anybody for use; and the updateability state determines what operations can be performed on the database. You control each of these aspects by using the *ALTER DATABASE* command to enable an option for the database. None of the state options uses the keywords *ON* and *OFF* to control the state value.

SINGLE_USER | RESTRICTED_USER | MULTI_USER

The three options SINGLE_USER, RESTRICTED_USER, and MULTI_USER describe the user access property of a database. They are mutually exclusive; setting any one of them unsets

the others. To set one of these options for your database, you just use the option name. For example, to set the *AdventureWorks2008* database to single-user mode, use the following code:

```
ALTER DATABASE AdventureWorks2008 SET SINGLE_USER;
```

A database in SINGLE_USER mode can have only one connection at a time. A database in RESTRICTED_USER mode can have connections only from users who are considered "qualified"—those who are members of the *dbcreator* or *sysadmin* server role or the *db_owner* role for that database. The default for a database is MULTI_USER mode, which means anyone with a valid user name in the database can connect to it. If you attempt to change a database's state to a mode that is incompatible with the current conditions—for example, if you try to change the database to SINGLE_USER mode when other connections exist—the behavior of SQL Server is determined by the TERMINATION option you specify. I'll discuss termination options shortly.

To determine which user access value is set for a database, you can examine the *sys.databases* catalog view, as shown here:

```
SELECT USER_ACCESS_DESC FROM sys.databases
WHERE name = '<name of database>';
```

This query will return one of MULTI_USER, SINGLE_USER, or RESTRICTED_USER.

OFFLINE | ONLINE | EMERGENCY

You use the OFFLINE, ONLINE, and EMERGENCY options to describe the status of a database. They are mutually exclusive. The default for a database is ONLINE. As with the user access options, when you use *ALTER DATABASE* to put the database in one of these modes, you don't specify a value of ON or OFF—you just use the name of the option. When a database is set to OFFLINE, it is closed and shut down cleanly and marked as offline. The database cannot be modified while the database is offline. A database cannot be put into OFFLINE mode if there are any connections in the database. Whether SQL Server waits for the other connections to terminate or generates an error message is determined by the TERMINATION option specified.

The following code examples show how to set a database's status value to OFFLINE and how to determine the status of a database:

```
ALTER DATABASE AdventureWorks2008 SET OFFLINE;
SELECT state_desc from sys.databases
WHERE name = 'AdventureWorks2008';
```

A database can be explicitly set to EMERGENCY mode, and that option will be discussed in Chapter 11, in conjunction with DBCC commands.

As shown in the preceding query, you can determine the current status of a database by examining the *state_desc* column of the *sys.databases* view. This column can return status

values other than OFFLINE, ONLINE, and EMERGENCY, but those values are not directly settable using *ALTER DATABASE*. A database can have the status value RESTORING while it is in the process of being restored from a backup. It can have the status value RECOVERING during a restart of SQL Server. The recovery process is performed on one database at a time, and until SQL Server has finished recovering a database, the database has a status of RECOVERING. If the recovery process cannot be completed for some reason (most likely because one or more of the log files for the database is unavailable or unreadable), SQL Server gives the database the status of RECOVERY_PENDING. Your databases can also be put into RECOVERY_PENDING mode if SQL Server runs out of either log or data space during rollback recovery, or if SQL Server runs out of locks or memory during any part of the startup process. I'll go into more detail about the difference between rollback recovery and startup recovery in Chapter 4.

If all the needed resources, including the log files, are available, but corruption is detected during recovery, the database may be put in the SUSPECT state. You can determine the state value by looking at the *state_desc* column in the *sys.databases* view. A database is completely unavailable if it's in the SUSPECT state, and you will not even see the database listed if you run *sp_helpdb*. However, you can still see the status of a suspect database in the *sys.databases* view. In many cases, you can make a suspect database available for read-only operations by setting its status to EMERGENCY mode. If you really have lost one or more of the log files for a database, EMERGENCY mode allows you to access the data while you copy it to a new location. When you move from RECOVERY_PENDING to EMERGENCY, SQL Server shuts down the database and then restarts it with a special flag that allows it to skip the recovery process. Skipping recovery can mean you have logically or physically inconsistent data—missing index rows, broken page links, or incorrect metadata pointers. By specifically putting your database in EMERGENCY mode, you are acknowledging that the data might be inconsistent but that you want access to it anyway.

READ_ONLY | READ_WRITE

These options describe the updatability of a database. They are mutually exclusive. The default for a database is READ_WRITE. As with the user access options, when you use *ALTER DATABASE* to put the database in one of these modes, you don't specify a value of ON or OFF, you just use the name of the option. When the database is in READ_WRITE mode, any user with the appropriate permissions can carry out data modification operations. In READ_ONLY mode, no INSERT, UPDATE, or DELETE operations can be executed. In addition, because no modifications are done when a database is in READ_ONLY mode, automatic recovery is not run on this database when SQL Server is restarted, and no locks need to be acquired during any *SELECT* operations. Shrinking a database in READ_ONLY mode is not possible.

A database cannot be put into READ_ONLY mode if there are any connections to the database. Whether SQL Server waits for the other connections to terminate or generates an error message is determined by the TERMINATION option specified.

The following code shows how to set a database's updatability value to READ_ONLY and how to determine the updatability of a database:

```
ALTER DATABASE AdventureWorks2008 SET READ_ONLY;
SELECT name, is_read_only FROM sys.databases
WHERE name = 'AdventureWorks2008';
```

When READ_ONLY is enabled for database, the *is_read_only* column returns 1; otherwise, for a READ_WRITE database, it returns 0.

Termination Options

As I just mentioned, several of the state options cannot be set when a database is in use or when it is in use by an unqualified user. You can specify how SQL Server should handle this situation by indicating a termination option in the *ALTER DATABASE* command. You can have SQL Server wait for the situation to change, generate an error message, or terminate the connections of unqualified users. The termination option determines the behavior of SQL Server in the following situations:

- When you attempt to change a database to SINGLE_USER and it has more than one current connection
- When you attempt to change a database to RESTRICTED_USER and unqualified users are currently connected to it
- When you attempt to change a database to OFFLINE and there are current connections to it
- When you attempt to change a database to READ_ONLY and there are current connections to it

The default behavior of SQL Server in any of these situations is to wait indefinitely. The following TERMINATION options change this behavior:

- **ROLLBACK AFTER integer [SECONDS]** This option causes SQL Server to wait for the specified number of seconds and then break unqualified connections. Incomplete transactions are rolled back. When the transition is to SINGLE_USER mode, all connections are unqualified except the one issuing the *ALTER DATABASE* command. When the transition is to RESTRICTED_USER mode, unqualified connections are those of users who are not members of the *db_owner* fixed database role or the *dbcreator* and *sysadmin* fixed server roles.
- **ROLLBACK IMMEDIATE** This option breaks unqualified connections immediately. All incomplete transactions are rolled back. Keep in mind that although the connection may be broken immediately, the rollback might take some time to complete. All work done by the transaction must be undone, so for certain operations, such as a batch update of millions of rows or a large index rebuild, you could be in for a long wait. Unqualified connections are the same as those described previously.

- **NO_WAIT** This option causes SQL Server to check for connections before attempting to change the database state and causes the *ALTER DATABASE* command to fail if certain connections exist. If the database is being set to SINGLE_USER mode, the *ALTER DATABASE* command fails if any other connections exist. If the transition is to RESTRICTED_USER mode, the *ALTER DATABASE* command fails if any unqualified connections exist.

The following command changes the user access option of the *AdventureWorks2008* database to SINGLE_USER and generates an error if any other connections to the *AdventureWorks2008* database exist:

```
ALTER DATABASE AdventureWorks2008 SET SINGLE_USER WITH NO_WAIT;
```

Cursor Options

The cursor options control the behavior of server-side cursors that were defined using one of the following T-SQL commands for defining and manipulating cursors: *DECLARE, OPEN, FETCH, CLOSE,* and *DEALLOCATE*.

- **CURSOR_CLOSE_ON_COMMIT {ON | OFF}** When this option is set to ON, any open cursors are closed (in compliance with SQL-92) when a transaction is committed or rolled back. If OFF (the default) is specified, cursors remain open after a transaction is committed. Rolling back a transaction closes any cursors except those defined as *INSENSITIVE* or *STATIC*.
- **CURSOR_DEFAULT {LOCAL | GLOBAL}** When this option is set to LOCAL and cursors aren't specified as GLOBAL when they are created, the scope of any cursor is local to the batch, stored procedure, or trigger in which it was created. The cursor name is valid only within this scope. The cursor can be referenced by local cursor variables in the batch, stored procedure, or trigger, or by a stored procedure output parameter. When this option is set to GLOBAL and cursors aren't specified as LOCAL when they are created, the scope of the cursor is global to the connection. The cursor name can be referenced in any stored procedure or batch executed by the connection.

Auto Options

The auto options affect actions that SQL Server might take automatically. All these options are Boolean options, with a value of ON or OFF.

- **AUTO_CLOSE** When this option is set to ON, the database is closed and shut down cleanly when the last user of the database exits, thereby freeing any resources. All file handles are closed, and all in-memory structures are removed so that the database is not using any memory. When a user tries to use the database again, it reopens. If the database was shut down cleanly, the database isn't initialized (reopened) until a user

tries to use the database the next time SQL Server is restarted. The AUTO_CLOSE option is handy for personal SQL Server databases because it allows you to manage database files as normal files. You can move them, copy them to make backups, or even e-mail them to other users. However, you shouldn't use this option for databases accessed by an application that repeatedly makes and breaks connections to SQL Server. The overhead of closing and reopening the database between each connection will hurt performance.

- **AUTO_SHRINK** When this option is set to ON, all of a database's files are candidates for periodic shrinking. Both data files and log files can be automatically shrunk by SQL Server. The only way to free space in the log files so that they can be shrunk is to back up the transaction log or set the recovery model to SIMPLE. The log files shrink at the point that the log is backed up or truncated. This option is never recommended.
- **AUTO_CREATE_STATISTICS** When this option is set to ON (the default), the SQL Server Query Optimizer creates statistics on columns referenced in a query's WHERE or ON clause. Adding statistics improves query performance because the SQL Server Query Optimizer can better determine how to evaluate a query.
- **AUTO_UPDATE_STATISTICS** When this option is set to ON (the default), existing statistics are updated if the data in the tables has changed. SQL Server keeps a counter of the modifications made to a table and uses it to determine when statistics are outdated. When this option is set to OFF, existing statistics are not automatically updated. (They can be updated manually.) Statistics will be discussed in more detail in Chapter 6 and Chapter 8, "The Query Optimizer."

SQL Options

The SQL options control how various SQL statements are interpreted. They are all Boolean options. The default for all these options is OFF for SQL Server, but many tools, such as the Management Studio, and many programming interfaces, such as ODBC, enable certain session-level options that override the database options and make it appear as if the ON behavior is the default.

- **ANSI_NULL_DEFAULT** When this option is set to ON, columns comply with the ANSI SQL-92 rules for column nullability. That is, if you don't specifically indicate whether a column in a table allows NULL values, NULLs are allowed. When this option is set to OFF, newly created columns do not allow NULLs if no nullability constraint is specified.
- **ANSI_NULLS** When this option is set to ON, any comparisons with a NULL value result in UNKNOWN, as specified by the ANSI-92 standard. If this option is set to OFF, comparisons of non-Unicode values to NULL result in a value of TRUE if both values being compared are NULL.

- **ANSI_PADDING** When this option is set to ON, strings being compared with each other are set to the same length before the comparison takes place. When this option is OFF, no padding takes place.
- **ANSI_WARNINGS** When this option is set to ON, errors or warnings are issued when conditions such as division by zero or arithmetic overflow occur.
- **ARITHABORT** When this option is set to ON, a query is terminated when an arithmetic overflow or division-by-zero error is encountered during the execution of a query. When this option is OFF, the query returns NULL as the result of the operation.
- **CONCAT_NULL_YIELDS_NULL** When this option is set to ON, concatenating two strings results in a NULL string if either of the strings is NULL. When this option is set to OFF, a NULL string is treated as an empty (zero-length) string for the purposes of concatenation.
- **NUMERIC_ROUNDABORT** When this option is set to ON, an error is generated if an expression will result in loss of precision. When this option is OFF, the result is simply rounded. The setting of ARITHABORT determines the severity of the error. If ARITHABORT is OFF, only a warning is issued and the expression returns a NULL. If ARITHABORT is ON, an error is generated and no result is returned.
- **QUOTED_IDENTIFIER** When this option is set to ON, identifiers such as table and column names can be delimited by double quotation marks, and literals must then be delimited by single quotation marks. All strings delimited by double quotation marks are interpreted as object identifiers. Quoted identifiers don't have to follow the T-SQL rules for identifiers when QUOTED_IDENTIFIER is ON. They can be keywords and can include characters not normally allowed in T-SQL identifiers, such as spaces and dashes. You can't use double quotation marks to delimit literal string expressions; you must use single quotation marks. If a single quotation mark is part of the literal string, it can be represented by two single quotation marks (''). This option must be set to ON if reserved keywords are used for object names in the database. When it is OFF, identifiers can't be in quotation marks and must follow all T-SQL rules for identifiers.
- **RECURSIVE_TRIGGERS** When this option is set to ON, triggers can fire recursively, either directly or indirectly. Indirect recursion occurs when a trigger fires and performs an action that causes a trigger on another table to fire, thereby causing an update to occur on the original table, which causes the original trigger to fire again. For example, an application updates table *T1,* which causes trigger *Trig1* to fire. *Trig1* updates table *T2,* which causes trigger *Trig2* to fire. *Trig2* in turn updates table *T1,* which causes *Trig1* to fire again. Direct recursion occurs when a trigger fires and performs an action that causes the same trigger to fire again. For example, an application updates table *T3,* which causes trigger *Trig3* to fire. *Trig3* updates table *T3* again, which causes trigger *Trig3* to fire again. When this option is OFF (the default), triggers can't be fired recursively.

Database Recovery Options

The database option RECOVERY (FULL, BULK_LOGGED or SIMPLE) determines how much recovery can be done on a SQL Server database. It also controls how much information is logged and how much of the log is available for backups. I'll cover this option in more detail in Chapter 4.

Two other options also apply to work done when a database is recovered. Setting the TORN_PAGE_DETECTION option to ON or OFF is possible in SQL Server 2008, but that particular option will go away in a future version. The recommended alternative is to set the PAGE_VERIFY option to a value of TORN_PAGE_DETECTION or CHECKSUM. (So TORN_PAGE_DETECTION should now be considered a value, rather the name of an option.)

The PAGE_VERIFY options discover damaged database pages caused by disk I/O path errors, which can cause database corruption problems. The I/O errors themselves are generally caused by power failures or disk failures that occur when a page is being written to disk.

- **CHECKSUM** When the PAGE_VERIFY option is set to CHECKSUM, SQL Server calculates a checksum over the contents of each page and stores the value in the page header when a page is written to disk. When the page is read from disk, a checksum is recomputed and compared with the value stored in the page header. If the values do not match, error message 824 (indicating a checksum failure) is reported.
- **TORN_PAGE_DETECTION** When the PAGE_VERIFY option is set to TORN_PAGE_DETECTION, it causes a bit to be flipped for each 512-byte sector in a database page (8 KB) whenever the page is written to disk. It allows SQL Server to detect incomplete I/O operations caused by power failures or other system outages. If a bit is in the wrong state when the page is later read by SQL Server, it means that the page was written incorrectly. (A torn page has been detected.) Although SQL Server database pages are 8 KB, disks perform I/O operations using 512-byte sectors. Therefore, 16 sectors are written per database page. A torn page can occur if the system crashes (for example, because of power failure) between the time the operating system writes the first 512-byte sector to disk and the completion of the 8-KB I/O operation. When the page is read from disk, the torn bits stored in the page header are compared with the actual page sector information. Unmatched values indicate that only part of the page was written to disk. In this situation, error message 824 (indicating a torn page error) is reported. Torn pages are typically detected by database recovery if it is truly an incomplete write of a page. However, other I/O path failures can cause a torn page at any time.
- **NONE (No Page Verify Option)** You can specify that that neither the CHECKSUM nor the TORN_PAGE_DETECTION value will be generated when a page is written, and these values will not be verified when a page is read.

Both checksum and torn page errors generate error message 824, which is written to both the SQL Server error log and the Windows event log. For any page that generates an

824 error when read, SQL Server inserts a row into the system table *suspect_pages* in the *msdb* database. (*SQL Server Books Online* has more information on "Understanding and Managing the *suspect_pages* Table.")

SQL Server retries any read that fails with a checksum, torn page, or other I/O error four times. If the read is successful in any one of those attempts, a message is written to the error log and the command that triggered the read continues. If the attempts fail, the command fails with error message 824.

You can "fix" the error by restoring the data or potentially rebuilding the index if the failure is limited to index pages. If you encounter a checksum failure, you can run *DBCC CHECKDB* to determine the type of database page or pages affected. You should also determine the root cause of the error and correct the problem as soon as possible to prevent additional or ongoing errors. Finding the root cause requires investigating the hardware, firmware drivers, BIOS, filter drivers (such as virus software), and other I/O path components.

In SQL Server 2008 and SQL Server 2005, the default is CHECKSUM. In SQL Server 2000, TORN_PAGE_DETECTION was the default, and CHECKSUM was not available. If you upgrade a database from SQL Server 2000, the PAGE_VERIFY value will be NONE or TORN_PAGE_DETECTION. You should always consider using CHECKSUM. Although TORN_PAGE_DETECTION uses fewer resources, it provides less protection than CHECKSUM. Keep in mind that if you enable CHECKSUM on a database upgraded from SQL Server 2000, that a checksum value is computed only on pages that are modified.

Note Prior to SQL Server 2008, neither CHECKSUM nor TORN_PAGE_DETECTION was available in the *tempdb* database.

Other Database Options

Of the other categories of database options, two more will be covered in later chapters. The snapshot isolation options will be discussed in Chapter 10, "Transactions and Concurrency." and the change tracking options were covered in Chapter 2. The others are beyond the scope of this book.

Database Snapshots

An interesting feature added to the product in SQL Server 2005 Enterprise Edition is database snapshots, which allow you to create a point-in-time, read-only copy of any database. In fact, you can create multiple snapshots of the same source database at different points in time. The actual space needed for each snapshot is typically much less than the space required for the original database because the snapshot stores only pages that have changed, as will be discussed shortly.

Database snapshots allow you to do the following:

- Turn a database mirror into a reporting server. (You cannot read from a database mirror, but you can create a snapshot of the mirror and read from that.)
- Generate reports without blocking or being blocked by production operations.
- Protect against administrative or user errors.

You'll probably think of more ways to use snapshots as you gain experience working with them.

Creating a Database Snapshot

The mechanics of snapshot creation are straightforward—you simply specify an option for the *CREATE DATABASE* command. There is no graphical interface for creating a database snapshot through Object Explorer, so you must use the T-SQL syntax. When you create a snapshot, you must include each data file from the source database in the *CREATE DATABASE* command, with the original logical name and a new physical name and path. No other properties of the files can be specified, and no log file is used.

Here is the syntax to create a snapshot of the *AdventureWorks2008* database, putting the snapshot files in the SQL Server 2008 default data directory:

```
CREATE DATABASE AdventureWorks_snapshot ON
( NAME = N'AdventureWorks_Data',
  FILENAME =
 N'C:\Program Files\Microsoft SQL Server\MSSQL10.MSSQLSERVER\MSSQL\
   Data\AW_data_snapshot.mdf')
AS SNAPSHOT OF AdventureWorks2008;
```

Each file in the snapshot is created as a sparse file, which is a feature of the NTFS file system. (Don't confuse sparse files with sparse columns available in SQL Server 2008.) Initially, a sparse file contains no user data, and disk space for user data has not been allocated to it. As data is written to the sparse file, NTFS allocates disk space gradually. A sparse file can potentially grow very large. Sparse files grow in 64-KB increments; thus, the size of a sparse file on disk is always a multiple of 64 KB.

The snapshot files contain only the data that has changed from the source. For every file, SQL Server creates a bitmap that is kept in cache, with a bit for each page of the file, indicating whether that page has been copied to the snapshot. Every time a page in the source is updated, SQL Server checks the bitmap for the file to see if the page has already been copied, and if it hasn't, it is copied at that time. This operation is called a *copy-on-write operation.* Figure 3-4 shows a database with a snapshot that contains 10 percent of the data (one page) from the source.

When a process reads from the snapshot, it first accesses the bitmap to see whether the page it wants is in the snapshot file or is still the source. Figure 3-5 shows read operations from the same database as in Figure 3-4. Nine of the pages are accessed from the source database, and one is accessed from the snapshot because it has been updated on the source. When a process reads from a snapshot database, no locks are taken no matter what isolation level

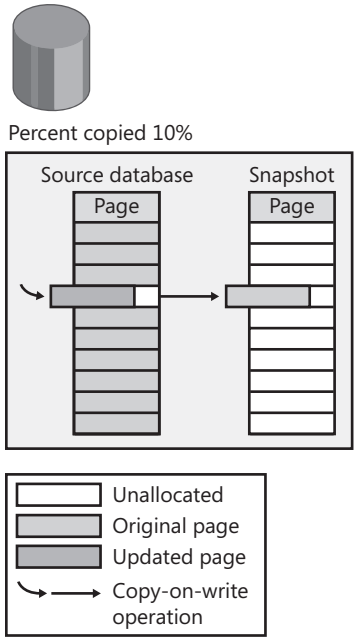


FIGURE 3-4 A database snapshot that contains one page of data from the source database

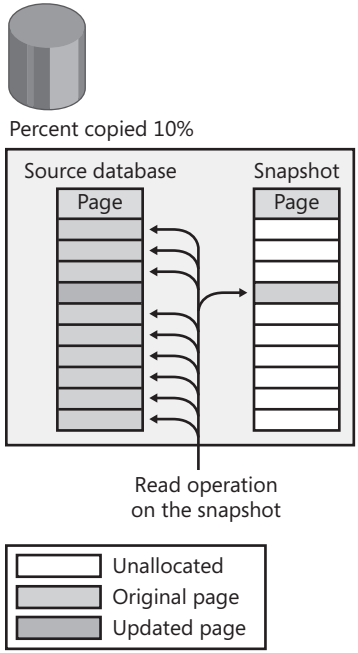


FIGURE 3-5 Read operations from a database snapshot, reading changed pages from the snapshot and unchanged pages from the source database

you are in. This is true whether the page is read from the sparse file or from the source database. This is one of the big advantages of using database snapshots.

As mentioned earlier, the bitmap is stored in cache, not with the file itself, so it is always readily available. When SQL Server shuts down or the database is closed, the bitmaps are lost and need to be reconstructed at database startup. SQL Server determines whether each page is in the sparse file as it is accessed, and then it records that information in the bitmap for future use.

The snapshot reflects the point in time when the *CREATE DATABASE* command is issued—that is, when the creation operation commences. SQL Server checkpoints the source database and records a synchronization Log Sequence Number (LSN) in the source database's transaction log. As you'll see in Chapter 4, when I talk about the transaction log, the LSN is a way to determine a specific point in time in a database. SQL Server then runs recovery on the source database so that any uncommitted transactions are rolled back in the snapshot. So although the sparse file for the snapshot starts out empty, it might not stay that way for long. If transactions are in progress at the time the snapshot is created, the recovery process has to undo uncommitted transactions before the snapshot database can be usable, so the snapshot contains the original versions of any page in the source that contains modified data.

Snapshots can be created only on NTFS volumes because they are the only volumes that support the sparse file technology. If you try to create a snapshot on a FAT or FAT32 volume, you'll get an error like one of the following:

```
Msg 1823, Level 16, State 2, Line 1
A database snapshot cannot be created because it failed to start.

Msg 5119, Level 16, State 1, Line 1
Cannot make the file "E:\AW_snapshot.MDF" a sparse file. Make sure the file system supports
sparse files.
```

The first error is basically the generic failure message, and the second message provides more details about why the operation failed.

Space Used by Database Snapshots

You can find out the number of bytes that each sparse file of the snapshot is currently using on disk by looking at the Dynamic Management Function *sys.dm_io_virtual_file_stats*, which returns the current number of bytes in a file in the *size_on_disk_bytes* column. This function takes *database_id* and *file_id* as parameters. The database ID of the snapshot database and the file IDs of each of its sparse files are displayed in the *sys.master_files* catalog view. You can also view the size in Windows Explorer by right-clicking the file name and looking at the properties, as shown in Figure 3-6. The Size value is the maximum size, and the size on disk should be the same value that you see using *sys.dm_io_virtual_file_stats*. The maximum size should be about the same size the source database was when the snapshot was created.

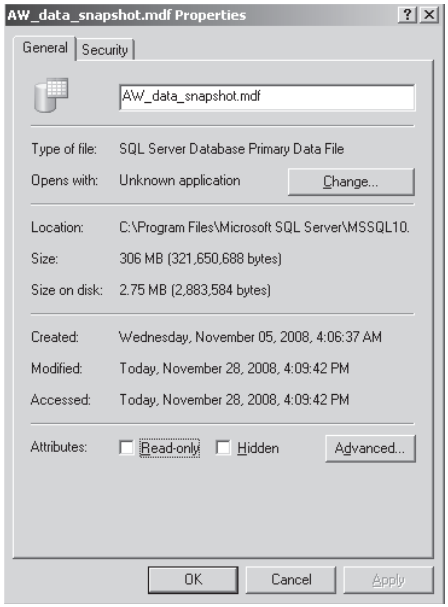


FIGURE 3-6 The snapshot file's Properties dialog box in Windows Explorer showing the current size of the sparse file as the size on disk

Because it is possible to have multiple snapshots for the same database, you need to make sure you have enough disk space available. The snapshots start out relatively small, but as the source database is updated, each snapshot grows. Allocations to sparse files are made in fragments called *regions,* in units of 64 KB. When a region is allocated, all the pages are zeroed out except the one page that has changed. There is then space for seven more changed pages in the same region, and a new region is not allocated until those seven pages are used.

It is possible to overcommit your storage. This means that under normal circumstances, you can have many times more snapshots than you have physical storage for, but if the snapshots grow, the physical volume might run out of space. (Note that this can happen when running online *DBCC CHECKDB,* and related commands, which use a hidden snapshot during processing. You have no control of the placement of the hidden snapshot that the commands use—they're placed on the same volume that the files of the parent database reside on. If this happens, the DBCC uses the source database and acquires table locks. You can read lots more details of the internals of the DBCC commands in Chapter 11.) Once the physical volume runs out of space, the write operations to the source cannot copy the Before image of the page to the sparse file. The snapshots that cannot write their pages out are marked as suspect and are unusable, but the source database continues operating normally. There is no way to "fix" a suspect snapshot; you must drop the snapshot database.

Managing Your Snapshots

If any snapshots exist on a source database, the source database cannot be dropped, detached, or restored. In addition, you can basically replace the source database with one of its snapshots by reverting the source database to the way it was when a snapshot was made. You do this by using the *RESTORE* command:

```
RESTORE DATABASE AdventureWorks2008
FROM DATABASE_SNAPSHOT = 'AdventureWorks_snapshot';
```

During the revert operation, both the snapshot and the source database are unavailable and are marked as "In restore." If an error occurs during the revert operation, the operation tries to finish reverting when the database starts again. You cannot revert to a snapshot if multiple snapshots exist, so you should first drop all snapshots except the one you want to revert to. Dropping a snapshot is like using any other *DROP DATABASE* operation. When the snapshot is deleted, all the NTFS sparse files are also deleted.

Keep in mind these additional considerations regarding database snapshots:

- Snapshots cannot be created for the *model*, *master*, or *tempdb* database. (Internally, snapshots can be created to run the online DBCC checks on the *master* database, but they cannot be created explicitly.)
- A snapshot inherits the security constraints of its source database, and because it is read-only, you cannot change the permissions.
- If you drop a user from the source database, the user is still in the snapshot.
- Snapshots cannot be backed up or restored, but backing up the source database works normally; it is unaffected by database snapshots.
- Snapshots cannot be attached or detached.
- Full-text indexing is not supported on database snapshots, and full-text catalogs are not propagated from the source database.

The *tempdb* Database

In some ways, the *tempdb* database is just like any other database, but it has some unique behaviors. Not all of them are relevant to the topic of this chapter, so I will provide some references to other chapters where you can find additional information.

As mentioned previously, the biggest difference between *tempdb* and all the other databases in your SQL Server instance is that *tempdb* is re-created—not recovered—every time SQL Server is restarted. You can think of *tempdb* as a workspace for temporary user objects and internal objects explicitly created by SQL Server itself.

Every time *tempdb* is re-created, it inherits most database options from the model database. However, the recovery model is not copied because *tempdb* always uses simple recovery,

which will be discussed in detail in Chapter 4. Certain database options cannot be set for *tempdb*, such as OFFLINE and READONLY. You also cannot drop the *tempdb* database.

In the SIMPLE recovery model, the *tempdb* database's log is constantly being truncated, and it can never be backed up. No recovery information is needed because every time SQL Server is started, *tempdb* is completely re-created; any previous user-created temporary objects (that is, all your tables and data) disappear.

Logging for *tempdb* is also different than for other databases. (Normal logging will be discussed in Chapter 4.) Many people assume that there is no logging in *tempdb*, but this is not true. Operations within *tempdb* are logged so that transactions on temporary objects can be rolled back, but the records in the log contain only enough information to roll back a transaction, not to recover (or redo) it.

As I mentioned previously, recovery is run on a database as one of the first steps in creating a snapshot. We can't recover *tempdb*, so we cannot create a snapshot of it, and this means we can't run *DBCC CHECKDB* using a snapshot (or, in fact, most of the DBCC validation commands). Another difference with running DBCC in *tempdb* is that SQL Server skips all allocation and catalog checks. Running *DBCC CHECKDB* (or *CHECKTABLE*) in *tempdb* acquires a Shared Table lock on each table as it is checked. (Locking will be discussed in Chapter 10.)

Objects in *tempdb*

Three types of objects are stored in *tempdb*: user objects, internal objects, and the version store, used primarily for snapshot isolation.

User Objects

All users have the privileges to create and use local and global temporary tables that reside in *tempdb*. (Local and global table names have the # or ## prefix, respectively. However, by default, users don't have the privileges to use *tempdb* and then create a table there, unless the table name is prefaced with # or ##.) But you can easily grant the privileges in an autostart procedure that runs each time SQL Server is restarted.

Other user objects that need space in *tempdb* include table variables and table-valued functions. The user objects that are created in *tempdb* are in many ways treated just like user objects in any other database. Space must be allocated for them when they are populated, and the metadata needs to be managed. You can see user objects by examining the system catalog views, such as *sys.objects*, and information in the *sys.partitions* and *sys.allocation_units* views will allow you to see how much space is taken up by user objects. I'll discuss these views in Chapters 5 and 7.

Internal Objects

Internal objects in *tempdb* are not visible using the normal tools, but they still take up space from the database. They are not listed in the catalog views because their metadata is stored only in memory. The three basic types of internal objects are work tables, work files, and sort units.

Work tables are created by SQL Server during the following operations:

- Spooling, to hold intermediate results during a large query
- Running *DBCC CHECKDB* or *DBCC CHECKTABLE*
- Working with XML or *varchar(MAX)* variables
- Processing SQL Service Broker objects
- Working with static or keyset cursors

Work files are used when SQL Server is processing a query that uses a hash operator, either for joining or aggregating data.

Sort units are created when a sort operation takes place, and this occurs in many situations in addition to a query containing an ORDER BY clause. SQL Server uses sorting to build an index, and it might use sorting to process queries involving grouping. Certain types of joins might require that SQL Server sort the data before performing the join. Sort units are created in *tempdb* to hold the data as it is being sorted. SQL Server can also create sort units in user databases in addition to *tempdb*, in particular when creating indexes. As you'll see in Chapter 6, when you create an index, you have the option to do the sort in the current user database or in *tempdb*.

Version Store

The version store supports technology for row-level versioning of data. Older versions of updated rows are kept in *tempdb* in the following situations:

- When an AFTER trigger is fired
- When a Data Modification Language (DML) command is executed in a database that allows snapshot transactions
- When multiple active result sets (MARS) are invoked from a client application
- During online index builds or rebuilds when there is concurrent DML on the index

Versioning and snapshot transactions are discussed in detail in Chapter 10.

Optimizations in *tempdb*

Because *tempdb* is used for many more internal operations in SQL Server 2008 than in previous versions, you have to take care in monitoring and managing it. The next section presents some best practices and monitoring suggestions. In this section, I tell you about some of the internal optimizations in SQL Server that allow *tempdb* to manage objects much more efficiently.

Logging Optimizations

As you know, every operation that affects your user database in any way is logged. In *tempdb*, however, this is not entirely true. For example, with logging update operations, only the original data (the "before" image) is logged, not the new values (the after image). In addition, the commit operations and committed log records are not flushed to disk synchronously in *tempdb*, as they are in other databases.

Allocation and Caching Optimizations

Many of the allocation optimizations are used in all databases, not just *tempdb*. However, *tempdb* is most likely the database in which the greatest number of new objects are created and dropped during production operations, so the impact on *tempdb* is greater than on user databases. In SQL Server 2008, allocation pages are accessed very efficiently to determine where free extents are available; you should see far less contention on the allocation pages than in previous versions. SQL Server 2008 also has a very efficient search algorithm for finding an available single page from mixed extents. When a database has multiple files, SQL Server 2008 has a very efficient proportional fill algorithm that allocates space to multiple data files, proportional to the amount of free space available in each file.

Another optimization specific to *tempdb* prevents you from having to allocate any new space for some objects. If a work table is dropped, one IAM page and one extent are saved (for a total of nine pages), so there is no need to deallocate and then reallocate the space if the same work table needs to be created again. This dropped work table cache is not very big and has room for only 64 objects. If a work table is truncated internally and the query plan that uses that worktable is still in the plan cache, again the first IAM page and the first extent are saved. For these truncated tables, there is no specific limitation on the number of objects that can be cached; it depends only on the available memory space.

User objects in *tempdb* can also have some of their space cached if they are dropped. For a small table of less than 8 MB, dropping a user object in *tempdb* causes one IAM page and one extent to be saved. However, if the table has had any additional DDL performed, such as creating indexes or constraints, or if the table was created using dynamic SQL, no caching is done.

For a large table, the entire drop is performed as a deferred operation. Deferred drop operations are in fact used in every database as a way to improve overall throughput because a thread does not need to wait for the drop to complete before proceeding with its next task. Like the other allocation optimizations that are available in all databases, the deferred drop probably provides the most benefit in *tempdb*, which is where tables are most likely to be dropped during production operations. A background thread eventually cleans up the space allocated for dropped tables, but until then, the allocated space remains. You can detect this space by looking at the *sys.allocation_units* system view for rows with a *type* value of 0, which indicates a dropped object; you will also see that the column called

container_id is 0, which indicates that the allocated space does not really belong to any object. I'll look at *sys.allocation_units* and the other system views that keep track of space usage in Chapter 5.

Best Practices

By default, your *tempdb* database is created on only one data file. You will probably find that multiple files give you better I/O performance and less contention on the global allocation structures (the GAM, SGAM, and PFS pages). An initial recommendation is that you have one file per CPU, but your own testing based on your data and usage patterns might indicate more or less than that. For the greatest efficiency with the proportional fill algorithm, the files should be the same size. The downside of multiple files is that every object will have multiple IAM pages and there will be more switching costs as objects are accessed. It will also take more effort just to manage the files. No matter how many files you have, they should be on the fastest disks you can afford. One log file should be sufficient, and that should also be on a fast disk.

To determine the optimum size of your *tempdb*, you must test your own applications with your data volumes, but knowing when and how *tempdb* is used can help you make preliminary estimates. Keep in mind that there is only one *tempdb* for each SQL Server instance, so one badly behaving application can affect all other users in all other applications. In Chapter 10, I'll explain how to determine the size of the version store. All these factors affect the space needed for your *tempdb*. Finally, in Chapter 11, I'll look at how the DBCC consistency checking commands use *tempdb* and how to determine the *tempdb* space requirements.

Database options for *tempdb* should rarely be changed, and some options are not applicable to *tempdb*. In particular, the autoshrink option is ignored in *tempdb*. In any case, shrinking *tempdb* is not recommended unless your workload patterns have changed significantly. If you do need to shrink your *tempdb*, you're probably better off shrinking each file individually. Keep in mind that the files might not be able to shrink if any internal objects or version store pages need to be moved. The best way to shrink *tempdb* is to ALTER the database, change the files' sizes, and then stop and restart SQL Server so *tempdb* is rebuilt to the desired size. You should allow your *tempdb* files to autogrow only as a last resort and only to prevent errors due to running out of room. You should not rely on autogrow to manage the size of your *tempdb* files. Autogrow causes a delay in processing when you can probably least afford it, although the impact is somewhat less if you use instant file initialization. You should determine the size of *tempdb* through testing and planning so that *tempdb* can start with as much space as it needs and won't have to grow while your applications are running.

Here are some tips for making optimum use of your *tempdb*. Later chapters will elaborate on why these suggestions are considered best practices:

- Take advantage of *tempdb* object caching.
- Keep your transactions short, especially those that use snapshot isolation, MARS, or triggers.

- If you expect a lot of allocation page contention, force a query plan that uses *tempdb* less.
- Avoid page allocation and deallocation by keeping columns that are to be updated at a fixed size rather than a variable size (which can implement the *UPDATE* as a *DELETE* followed by an *INSERT*).
- Do not mix long and short transactions from different databases (in the same instance) if versioning is being used.

tempdb Space Monitoring

Quite a few tools, stored procedures, and system views report on object space usage, as discussed in Chapters 5 and 7. However, one set of system views reports information only for *tempdb*. The simplest view is *sys.dm_db_file_space_usage*, which returns one row for each data file in *tempdb*. It returns the following columns:

- *database_id* (even though the *DBID* 2 is the only one used)
- *file_id*
- *unallocated_extent_page_count*
- *version_store_reserved_page_count*
- *user_object_reserved_page_count*
- *internal_object_reserved_page_count*
- *mixed_extent_page_count*

These columns can show you how the space in *tempdb* is being used for the three types of storage: user objects, internals objects, and version store.

Two other system views are similar to each other:

- ***sys.dm_db_task_space_usage*** This view returns one row for each active task and shows the space allocated and deallocated by the task for user objects and internal objects. If no tasks are being run by a session, this view still gives you one row for the session, with all the space values showing 0. No version store information is reported because that space is not associated with any particular task or session. Every running task starts with zeros for all the space allocation and deallocation values.
- ***sys.dm_db_session_space_usage*** This view returns one row for each session, with the cumulative values for space allocated and deallocated by the session for user objects and internal objects, for all tasks that have been completed. In general, the space allocated values should be the same as the space deallocated values, but if there are deferred drop operations, allocated values will be greater than the deallocated values. Keep in mind that this information is not available to all users; a special permission called *VIEW SERVER STATE* is needed to select from this view.

Database Security

Security is a huge topic that affects almost every action of every SQL Server user, including administrators and developers, and it deserves an entire book of its own. However, some areas of the SQL Server security framework are crucial to understanding how to work with a database or with any objects in a SQL Server database, so I can't leave the topic completely untouched in this book.

SQL Server manages a hierarchical collection of entities. The most prominent of these entities are the server and databases in the server. Underneath the database level are objects. Each of these entities below the server level is owned by individuals or groups of individuals. The SQL Server security framework controls access to the entities within a SQL Server instance. Like any resource manager, the SQL Server security model has two parts: authentication and authorization.

Authentication is the process by which the SQL Server validates and establishes the identity of an individual who wants to access a resource. *Authorization* is the process by which SQL Server decides whether a given identity is allowed to access a resource.

In this section, I'll discuss the basic issues of database access and then describe the metadata where information on database access is stored. I'll also tell you about the concept of schemas and describe how they are used to access objects.

The following two terms now form the foundation for describing security control in SQL Server 2008:

- **Securable** A *securable* is an entity on which permissions can be granted. Securables include databases, schemas, and objects.
- **Principal** A *principal* is an entity that can access securables. A *primary principal* represents a single user (such as a SQL Server login or a Windows login); a *secondary principal* represents multiple users (such as a role or a Windows group).

Database Access

Authentication is performed at two different levels in SQL Server. First, anyone who wants to access any SQL Server resource must be authenticated at the server level. SQL Server 2008 security provides two basic methods for authenticating logins: Windows Authentication and SQL Server Authentication. In Windows Authentication, SQL Server login security is integrated directly with Windows security, allowing the operating system to authenticate SQL Server users. In SQL Server Authentication, an administrator creates SQL Server login accounts within SQL Server, and any user connecting to SQL Server must supply a valid SQL Server login name and password.

Windows Authentication uses *trusted connections,* which rely on the impersonation feature of Windows. Through impersonation, SQL Server can take on the security context of the Windows user account initiating the connection and test whether the SID has a valid privilege level. Windows impersonation and trusted connections are supported by any of the available network libraries when connecting to SQL Server.

Under Windows Server 2003 and Windows Server 2008, SQL Server can use Kerberos to support mutual authentication between the client and the server, as well as to pass a client's security credentials between computers so that work on a remote server can proceed using the credentials of the impersonated client. With Windows Server 2003 and Windows Server 2008, SQL Server uses Kerberos and delegation to support Windows authentication as well as SQL Server authentication.

The authentication method (or methods) used by SQL Server is determined by its security mode. SQL Server can run in one of two security modes: Windows Authentication mode (which uses only Windows authentication) and Mixed mode (which can use either Windows authentication or SQL Server authentication, as chosen by the client). When you connect to an instance of SQL Server configured for Windows Authentication mode, you cannot supply a SQL Server login name, and your Windows user name determines your level of access to SQL Server.

One advantage of Windows authentication has always been that it allows SQL Server to take advantage of the security features of the operating system, such as password encryption, password aging, and minimum and maximum length restrictions on passwords. When running on Windows Server 2003 or Windows Server 2008, SQL Server authentication can also take advantage of Windows password policies. Take a look at the *ALTER LOGIN* command in *SQL Server Books Online* for the full details. Also note that if you choose Windows Authentication during setup, the default SQL Server *sa* login is disabled. If you switch to Mixed mode after setup, you can enable the *sa* login using the *ALTER LOGIN* command. You can change the authentication mode in Management Studio by right-clicking on the server name, choosing Properties, and then selecting the Security page. Under Server authentication, select the new server authentication mode, as shown in Figure 3-7.

Under Mixed mode, Windows-based clients can connect using Windows authentication, and connections that don't come from Windows clients or that come across the Internet can connect using SQL Server authentication. In addition, when a user connects to an instance of SQL Server that has been installed in Mixed mode, the connection can always supply a SQL Server login name explicitly. This allows a connection to be made using a login name distinct from the user name in Windows.

All login names, whether from Windows or SQL Server authentication, can be seen in the *sys.server_principals* catalog view, which also contains a SID for each server principal. If the principal is a Windows login, the SID is the same one that Windows uses to validate the user's access to Windows resources. The view contains rows for server roles, Windows groups, and logins mapped to certificates and asymmetric keys, but I will not discuss those principals here.

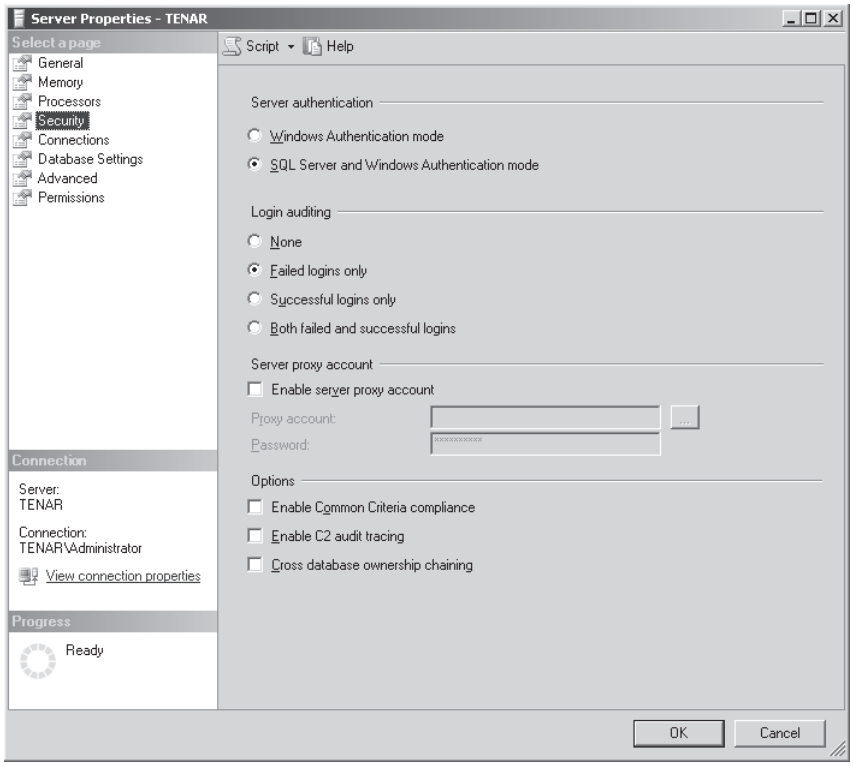


FIGURE 3-7 Choosing an authentication mode for your SQL Server instance in the Server Properties dialog box

Managing Database Security

Login names can be the owners of databases, as seen in the *sys.databases* view, which has a column for the SID of the login that owns the database. Databases are the only resource owned by login names. As you'll see, all objects within a database are owned by database principals.

The SID used by a principal determines which databases that principal has access to. Each database has a *sys.database_principals* catalog view, which you can think of as a mapping table that maps login names to users in that particular database. Although a login name and a user name can have the same value, they are separate things. The following query shows the mapping of users in the *AdventureWorks2008* database to login names, and it also shows the default schema (which I will discuss shortly) for each database user:

```
SELECT s.name as [Login Name], d.name as [User Name],
     default_schema_name as [Default Schema]
   FROM sys.server_principals s
      JOIN sys.database_principals d
   ON d.sid = s.sid;
```

In my *AdventureWorks2008* database, these are the results I receive:

```
Login Name User Name  Default Schema
---------- ---------- --------------
sa         dbo        dbo
sue        sue        sue
```

Note that the login *sue* has the same value for the user name in this database. There is no guarantee that other databases that *sue* has access to will use the same user name. The dbo is a special database principal that is mapped to the login owning the database and it is also used by all logins in the sysadmin server role. Within a database, it is users, not logins, who own objects, and users, not logins, to whom permissions are granted.

The preceding results also indicate the default schema for each user in my *AdventureWorks2008* database. In this case, the default schema is the same as the user name, but that doesn't have to be the case, as you'll see in the next section.

Databases vs. Schemas

In the ANSI SQL-92 standard, a *schema* is defined as a collection of database objects that are owned by a single user and form a single namespace. A *namespace* is a set of objects that cannot have duplicate names. For example, two tables can have the same name only if they are in separate schemas, so no two tables in the same schema can have the same name. You can think of a schema as a container of objects. (In the context of database tools, a schema also refers to the catalog information that describes the objects in a schema or database. In SQL Server Analysis Services, a schema is a description of multidimensional objects such as cubes and dimensions.)

Principals and Schemas

Prior to SQL Server 2005, there was a *CREATE SCHEMA* command, but it effectively did nothing because there was an implicit relationship between users and schemas that could not be changed or removed. In fact, the relationship was so close that many users of these earlier versions of SQL Server were unaware that users and schemas are different things. Every user was the owner of a schema that has the same name as the user. If you created a user *sue*, for example, SQL Server 2000 created a schema called *sue*, which was *sue*'s default schema.

In SQL Server 2005 and SQL Server 2008, users and schemas are two separate things. To understand the difference between users and schemas, think of the following: Permissions are granted to users, but objects are placed in schemas.

The command *GRANT CREATE TABLE TO sue* refers to the user *sue*. Let's say *sue* then creates a table, as follows:

```
CREATE TABLE mytable (col1 varchar(20));
```

This table is placed in *sue*'s default schema, which may be the schema *sue*. If another user wants to retrieve data from this table, he can issue this statement:

```
SELECT col1 FROM sue.mytable;
```

In this statement, *sue* refers to the schema that contains the table.

Schemas can be owned by either primary or secondary principals. Although every object in a SQL Server 2008 database is owned by a user, you never reference an object by its owner; you reference it by the schema in which it is contained. In most cases, the owner of the schema is the same as the owner of all objects within the schema. The metadata view *sys.objects* contains a column called *principal_id*, which contains the *user_id* of an object's owner if it is not the same as the owner of the object's schema. In addition, a user is never added to a schema; schemas contain objects, not users. For backward compatibility, if you execute the *sp_adduser* or *sp_grantdbaccess* procedure to add a user to a database, SQL Server 2008 creates both a user and a schema of the same name, and it makes the schema the default schema for the new user. However, you should get used to using the new *DDL CREATE USER* and *CREATE SCHEMA* commands because *sp_adduser* and *sp_grantdbaccess* have been deprecated. When you create a user, you can specify a default schema if you want, but the default for the default schema is the *dbo* schema.

Default Schemas

When you create a new database in SQL Server 2008, several schemas are included in it. These include *dbo*, *INFORMATION_SCHEMA*, and *guest*. In addition, every database has a schema called *sys*, which provides a way to access all the system tables and views. Finally, every fixed database role except *public* has a schema of the same name in SQL Server 2008.

Users can be assigned a default schema that might or might not exist when the user is created. A user can have at most one default schema at any time. As mentioned earlier, if no default schema is specified for a user, the default schema for the user is *dbo*. A user's default schema is used for name resolution during object creation or object reference. This can be both good news and bad news for backward compatibility. The good news is that if you've upgraded a database from SQL Server 2000, which has many objects in the *dbo* schema, your code can continue to reference those objects without having to specify the schema explicitly. The bad news is that for object creation, SQL Server tries to create the object in the *dbo* schema rather than in a schema owned by the user creating the table. The user might not have permission to create objects in the *dbo* schema, even if that is the user's default schema. To avoid confusion, in SQL Server 2008 you should always specify the schema name for all object access as well as object management.

Note When a login in the *sysadmin* role creates an object with a single part name, the schema is always *dbo*. However, a *sysadmin* can explicitly specify an alternate schema in which to create an object.

To create an object in a schema, you must satisfy the following conditions:

- The schema must exist.
- The user creating the object must have permission to create the object (through *CREATE TABLE, CREATE VIEW, CREATE PROCEDURE,* and so on), either directly or through role membership.
- The user creating the object must be the owner of the schema or a member of the role that owns the schema, or the user must have ALTER rights on the schema or have the ALTER ANY SCHEMA permission in the database.

Moving or Copying a Database

You might need to move a database before performing maintenance on your system, after a hardware failure, or when you replace your hardware with a newer, faster system. Copying a database is a common way to create a secondary development or testing environment. You can move or copy a database by using a technique called *detach and attach* or by backing up the database and restoring it in the new location.

Detaching and Reattaching a Database

You can detach a database from a server by using a simple stored procedure. Detaching a database requires that no one is using the database. If you find existing connections that you can't terminate, you can use the *ALTER DATABASE* command and set the database to SINGLE_USER mode using one of the termination options that breaks existing connections. Detaching a database ensures that no incomplete transactions are in the database and that there are no dirty pages for this database in memory. If these conditions cannot be met, the detach operation fails. Once the database is detached, the entry for it is removed from the *sys.databases* catalog view and from the underlying system tables.

Here is the command to detach a database:

```
EXEC sp_detach_db <name of database>;
```

Once the database has been detached, from the perspective of SQL Server, it's as if you had dropped the database. No metadata for the database remains within the SQL Server instance, and the only time there might be a trace of it is when your *msdb* database contains backup and restore history for the database that has not yet been deleted. But the history of when backups and restores were done would provide no information about the structure or content of the database. If you are planning to reattach the database later, it's a good idea to record the properties of all the files that were part of the database.

Note The *DROP DATABASE* command also removes all traces of the database from your instance, but dropping a database is more severe than detaching. SQL Server makes sure that no one is connected to the database before dropping it, but it doesn't check for dirty pages or open transactions. Dropping a database also removes the physical files from the operating system, so unless you have a backup, the database is really gone.

To attach a database, you can use the *CREATE DATABASE* command with the FOR ATTACH option. (There is a stored procedure, *sp_attach_db*, but it is deprecated and not recommended in SQL Server 2008.) The *CREATE DATABASE* command gives you control over all the files and their placement and is not limited to only 16 files like *sp_attach_db* is. *CREATE DATABASE* has no such limit—in fact, you can specify up to 32,767 files and 32,767 file groups for each database. The syntax summary for the *CREATE DATABASE* command showing the attach options is shown here:

```
CREATE DATABASE database_name
    ON <filespec> [ ,...n ]
    FOR { ATTACH
        | ATTACH_REBUILD_LOG }
```

Note that only the primary file is required to have a <filespec> entry because the primary file contains information about the location of all the other files. If you'll be attaching existing files with a different path than when the database was first created or last attached, you must have additional <filespec> entries. In any event, all the data files for the database must be available, whether or not they are specified in the *CREATE DATABASE* command. If there are multiple log files, they must all be available.

However, if a read/write database has a single log file that is currently unavailable and if the database was shut down with no users or open transactions before the attach operation, FOR ATTACH rebuilds the log file and updates information about the log in the primary file. If the database is read-only, the primary file cannot be updated, so the log cannot be rebuilt. Therefore, when you attach a read-only database, you must specify the log file or files in the FOR ATTACH clause.

Alternatively, you can use the FOR ATTACH_REBUILD_LOG option, which specifies that the database will be created by attaching an existing set of operating system files. This option is limited to read/write databases. If one or more transaction log files are missing, the log is rebuilt. There must be a <filespec> entry specifying the primary file. In addition, if the log files are available, SQL Server uses those files instead of rebuilding the log files, so the FOR ATTACH_REBUILD_LOG will function as if you used FOR ATTACH.

If your transaction log is rebuilt by attaching the database, using the FOR ATTACH_REBUILD_LOG puts the database to SIMPLE recovery. If the database was originally in FULL or BULK_LOGGED recovery, it is recommended that you switch back to that original recovery model, and make a full backup after performing the ATTACH operation.

You typically use FOR ATTACH_REBUILD_LOG when you copy a read/write database with a large log to another server where the copy will be used mostly or exclusively for read operations and therefore require less log space than the original database.

Although the documentation says that you should use *CREATE DATABASE FOR ATTACH* only on databases that were previously detached using *sp_detach_db,* sometimes following this recommendation isn't necessary. If you shut down the SQL Server instance, the files are closed, just as if you had detached the database. However, you are not guaranteed that all dirty pages from the database were written to disk before the shutdown. This should not cause a problem when you attach such a database if the log file is available. The log file has a record of all completed transactions, and a full recovery is performed when the database is attached to make sure the database is consistent. One benefit of using the *sp_detach_db* procedure is that SQL Server records the fact that the database was shut down cleanly, and the log file does not have to be available to attach the database. SQL Server builds a new log file for you. This can be a quick way to shrink a log file that has become much larger than you would like, because the new log file that *sp_attach_db* creates for you would be the minimum size—less than 1 MB.

Backing Up and Restoring a Database

You can also use backup and restore to move a database to a new location, as an alternative to detach and attach. One benefit of this method is that the database does not need to come offline at all because backup is a completely online operation. Because this book is not a how-to book for database administrators, you should refer to the bibliography in the companion content for several excellent book recommendations about the mechanics of backing up and restoring a database and to learn best practices for setting up a backup-and-restore plan for your organization. Nevertheless, some issues relating to backup-and-restore processes can help you understand why one backup plan might be better suited to your needs than another, so I will discuss backup and restore briefly in Chapter 4. Most of these issues involve the role of the transaction log in backup-and-restore operations.

Moving System Databases

You might need to move system databases as part of a planned relocation or scheduled maintenance operation. If you move a system database and later rebuild the *master* database, you must move the system database again because the rebuild operation installs all system databases to their default location. The steps for moving *tempdb*, *model*, and *msdb* are slightly different than for moving the *master* database.

Note In SQL Server 2008, the *mssqlsystemresource* database cannot be moved. If you move the files for this database, you will not be able to restart your SQL Server service. This is incorrectly documented in the RTM edition of *SQL Server 2008 Books Online,* which indicates that the *mssqlsystemresource* database can be moved, but this misinformation may be corrected in a later refresh.

Here are the steps for moving an undamaged system database (that is, not the *master* database):

1. For each file in the database to be moved, use the *ALTER DATABASE* command with the MODIFY FILE option to specify the new physical location.
2. Stop the SQL Server instance.
3. Physically move the files.
4. Restart the SQL Server instance.
5. Verify the change by running the following query:

```
SELECT name, physical_name AS CurrentLocation, state_desc
FROM sys.master_files
WHERE database_id = DB_ID(N'<database_name>');
```

If the system database needs to be moved because of a hardware failure, the solution is a bit more problematical because you might not have access to the server to run the *ALTER DATABASE* command. Here are the steps to move a damaged system database (other than the *master* database or the resource database):

1. Stop the instance of SQL Server if it has been started.
2. Start the instance of SQL Server in *master-only* recovery mode (by specifying traceflag 3608) by entering one of the following commands at the command prompt:

```
-- If the instance is the default instance:
NET START MSSQLSERVER /f /T3608

-- For a named instance:
NET START MSSQL$instancename /f /T3608
```

3. For each file in the database to be moved, use the *ALTER DATABASE* command with the MODIFY FILE option to specify the new physical location. You can use either Management Studio or the SQLCMD utility.
4. Exit Management Studio or the SQLCMD utility.
5. Stop the instance of SQL Server.
6. Physically move the file or files to the new location.
7. Restart the instance of SQL Server without traceflag 3608. For example, run *NET START MSSQLSERVER*.
8. Verify the change by running the following query:

```
SELECT name, physical_name AS CurrentLocation, state_desc
FROM sys.master_files
WHERE database_id = DB_ID(N'<database_name>');
```

Moving the *master* Database

Full details on moving the *master* database can be found in *SQL Server Books Online,* but I will summarize the steps here. The biggest difference between moving this database and moving other system databases is that you must go through the SQL Server Configuration Manager.

To move the *master* database, follow these steps.

1. Open the SQL Server Configuration Manager. Right-click the desired instance of SQL Server, choose Properties, and then click the Advanced tab.
2. Edit the Startup Parameters values to point to the new directory location for the *master* database data and log files. If you want, you can also move the SQL Server error log files. The parameter value for the data file must follow the *–d* parameter, the value for the log file must follow the *–l* parameter, and the value for the error log must follow the *–e* parameter, as shown here:

```
-dE:\SQLData\master.mdf;
-lE:\SQLData\mastlog.ldf;
-eE:\SQLData\LOG\ERRORLOG
```

3. Stop the instance of SQL Server and physically move the files to the new location.
4. Restart the instance of SQL Server.
5. Verify the file change for the *master* database by running the following query:

```
SELECT name, physical_name AS CurrentLocation, state_desc
FROM sys.master_files
WHERE database_id = DB_ID('master');
```

Compatibility Levels

Each new version of SQL Server includes a large number of new features, many of which require new keywords and also change certain behaviors that existed in earlier versions. To provide maximum backward compatibility, Microsoft allows you to set the compatibility level of a database running on a SQL Server 2008 instance to one of the following modes: 100, 90, or 80. All newly created databases in SQL Server 2008 have a compatibility level of 100 unless you change the level for the *model* database. A database that has been upgraded or attached from an older version has its compatibility level set to the version from which the database was upgraded.

All the examples and explanations in this book assume that you're using a database in 100 compatibility mode, unless otherwise noted. If you find that your SQL statements behave differently than the ones in the book, you should first verify that your database is in 100 compatibility mode by executing this command:

```
SELECT compatibility_level FROM sys.databases
WHERE name =  '<database name>';
```

To change to a different compatibility level, use the *ALTER DATABASE* command:

```
ALTER DATABASE <database name>
SET COMPATIBILITY_LEVEL =  <compatibility-level>;
```

Note The compatibility-level options are intended to provide a transition period while you're upgrading a database or an application to SQL Server 2008. I strongly suggest that you try to change your applications so that compatibility options are not needed. Microsoft doesn't guarantee that these options will continue to work in future versions of SQL Server.

Not all changes in behavior from older versions of SQL Server can be duplicated by changing the compatibility level. For the most part, the differences have to do with whether new reserved keywords and new syntax are recognized, and they do not affect how your queries are processed internally. For example, if you change to compatibility level 80, you don't make the system tables viewable or do away with schemas. But because the word *MERGE* is a new reserved keyword in SQL Server 2008 (compatibility level 100), by setting your compatibility level to 80 or 90, you can create a table called *MERGE* without using any special delimiters—or a table that you already have in a SQL Server 2005 database continues to be accessible if the database stays in the 90 compatibility level.

For a complete list of the behavioral differences between the compatibility levels and the new reserved keywords, see the documentation for *ALTER DATABASE Compatibility Level* in *SQL Server Books Online*.

Summary

A database is a collection of objects such as tables, views, and stored procedures. Although a typical SQL Server installation has many databases, it always includes the following three: *master*, *model*, and *tempdb*. An installation usually also includes *msdb,* but that database can be removed. (To remove *msdb* requires a special traceflag and is rarely recommended.) A SQL Server instance also includes the *mssqlsystemresource* database that cannot be seen using the normal tools. Every database has its own transaction log; integrity constraints among objects keep a database logically consistent.

Databases are stored in operating system files in a one-to-many relationship. Each database has at least one file for data and one file for the transaction log. You can increase and decrease the size of databases and their files easily, either manually or automatically.

Chapter 10

Transactions and Concurrency

Kalen Delaney

Concurrency can be defined as the ability of multiple processes to access or change shared data at the same time. The greater the number of concurrent user processes that can be active without interfering with each other, the greater the concurrency of the database system.

Concurrency is reduced when a process that is changing data prevents other processes from reading that data or when a process that is reading data prevents other processes from changing that data. I use the terms *reading* or *accessing* to describe the impact of using the *SELECT* statement on your data. Concurrency is also affected when multiple processes attempt to change the same data simultaneously and they cannot all succeed without sacrificing data consistency. I use the terms *modifying, changing,* or *writing* to describe the impact of using the *INSERT*, *UPDATE*, *MERGE*, or *DELETE* statements on your data. (Note that *MERGE* is a new data modification statement in SQL Server 2008, and you can think of it as a combination of *INSERT, UPDATE,* and *DELETE*.)

In general, database systems can take two approaches to managing concurrent data access: optimistic or pessimistic. Microsoft SQL Server 2008 supports both approaches. Pessimistic concurrency was the only concurrency model available before SQL Server 2005. As of SQL Server 2005, you specify which model to use by using two database options and a SET option called TRANSACTION ISOLATION LEVEL.

After I describe the basic differences between the two models, we look at the five possible isolation levels in SQL Server 2008, as well as the internals of how SQL Server controls concurrent access using each model. We look at how to control the isolation level, and we look at the metadata that shows you what SQL Server is doing.

Concurrency Models

In either concurrency model, a conflict can occur if two processes try to modify the same data at the same time. The difference between the two models lies in whether conflicts can be avoided before they occur or can be dealt with in some manner after they occur.

Pessimistic Concurrency

With pessimistic concurrency, the default behavior is for SQL Server to acquire locks to block access to data that another process is using. Pessimistic concurrency assumes that enough data modification operations are in the system that any given read operation is likely

affected by a data modification made by another user. In other words, the system behaves pessimistically and assumes that a conflict will occur. Pessimistic concurrency avoids conflicts by acquiring locks on data that is being read, so no other processes can modify that data. It also acquires locks on data being modified, so no other processes can access that data for either reading or modifying. In other words, readers block writers and writers block readers in a pessimistic concurrency environment.

Optimistic Concurrency

Optimistic concurrency assumes that there are sufficiently few conflicting data modification operations in the system that any single transaction is unlikely to modify data that another transaction is modifying. The default behavior of optimistic concurrency is to use row versioning to allow data readers to see the state of the data before the modification occurs. Older versions of data rows are saved, so a process reading data can see the data as it was when the process started reading and not be affected by any changes being made to that data. A process that modifies the data is unaffected by processes reading the data because the reader is accessing a saved version of the data rows. In other words, readers do not block writers and writers do not block readers. Writers can and will block writers, however, and this is what causes conflicts. SQL Server generates an error message when a conflict occurs, but it is up to the application to respond to that error.

Transaction Processing

No matter what concurrency model you're working with, an understanding of transactions is crucial. A transaction is the basic unit of work in SQL Server. Typically, it consists of several SQL commands that read and update the database, but the update is not considered final until a *COMMIT* command is issued (at least for an explicit transaction). In general, when I talk about a modification operation or a read operation, I am talking about the transaction that performs the data modification or the read, which is not necessarily a single SQL statement. When I say that writers will block readers, I mean that so long as the transaction that performed the write operation is active, no other process can read the modified data.

The concept of a transaction is fundamental to understanding concurrency control. The mechanics of transaction control from a programming perspective are beyond the scope of this book, but I discuss basic transaction properties. I also go into detail about the transaction isolation levels because that has a direct impact on how SQL Server manages the data being accessed in your transactions.

An implicit transaction is any individual *INSERT*, *UPDATE*, *DELETE,* or *MERGE* statement. (You can also consider *SELECT* statements to be implicit transactions, although SQL Server does not write to the log when *SELECT* statements are processed.) No matter how many rows are affected, the statement must exhibit all the ACID properties of a transaction, which I tell you

about in the next section. An explicit transaction is one whose beginning is marked with a *BEGIN TRAN* statement and whose end is marked by a *COMMIT TRAN* or *ROLLBACK TRAN* statement. Most of the examples I present use explicit transactions because it is the only way to show the state of SQL Server in the middle of a transaction. For example, many types of locks are held for only the duration of the transaction. I can begin a transaction, perform some operations, look around in the metadata to see what locks are being held, and then end the transaction. When the transaction ends, the locks are released; I can no longer look at them.

ACID Properties

Transaction processing guarantees the consistency and recoverability of SQL Server databases. It ensures that all transactions are performed as a single unit of work—even in the presence of a hardware or general system failure. Such transactions are referred to as having the ACID properties, with *ACID* standing for *atomicity, consistency, isolation,* and *durability.* In addition to guaranteeing that explicit multistatement transactions maintain the ACID properties, SQL Server guarantees that an implicit transaction also maintains the ACID properties.

Here's an example in pseudocode of an explicit ACID transaction:

```
BEGIN TRANSACTION DEBIT_CREDIT
Debit savings account $1000
Credit checking account $1000
COMMIT TRANSACTION DEBIT_CREDIT
```

Now let's take a closer look at each of the ACID properties.

Atomicity

SQL Server guarantees the atomicity of its transactions. *Atomicity* means that each transaction is treated as all or nothing—it either commits or aborts. If a transaction commits, all its effects remain. If it aborts, all its effects are undone. In the preceding DEBIT_CREDIT example, if the savings account debit is reflected in the database but the checking account credit isn't, funds essentially disappear from the database—that is, funds are subtracted from the savings account but never added to the checking account. If the reverse occurs (if the checking account is credited and the savings account is not debited), the customer's checking account mysteriously increases in value without a corresponding customer cash deposit or account transfer. Because of the atomicity feature of SQL Server, both the debit and credit must be completed or else neither event is completed.

Consistency

The consistency property ensures that a transaction won't allow the system to arrive at an incorrect logical state—the data must always be logically correct. Constraints and rules are honored even in the event of a system failure. In the DEBIT_CREDIT example, the logical rule

is that money can't be created or destroyed: a corresponding, counterbalancing entry must be made for each entry. (Consistency is implied by, and in most situations redundant with, atomicity, isolation, and durability.)

Isolation

Isolation separates concurrent transactions from the updates of other incomplete transactions. In the DEBIT_CREDIT example, another transaction can't see the work in progress while the transaction is being carried out. For example, if another transaction reads the balance of the savings account after the debit occurs, and then the DEBIT_CREDIT transaction is aborted, the other transaction is working from a balance that never logically existed.

SQL Server accomplishes isolation among transactions automatically. It locks data or creates row versions to allow multiple concurrent users to work with data while preventing side effects that can distort the results and make them different from what would be expected if users were to serialize their requests (that is, if requests were queued and serviced one at a time). This serializability feature is one of the isolation levels that SQL Server supports. SQL Server supports multiple isolation levels so that you can choose the appropriate tradeoff between how much data to lock, how long to hold locks, and whether to allow users access to prior versions of row data. This tradeoff is known as concurrency vs. consistency.

Durability

After a transaction commits, the durability property of SQL Server ensures that the effects of the transaction persist even if a system failure occurs. If a system failure occurs while a transaction is in progress, the transaction is completely undone, leaving no partial effects on the data. For example, if a power outage occurs in the middle of a transaction before the transaction is committed, the entire transaction is rolled back when the system is restarted. If the power fails immediately after the acknowledgment of the commit is sent to the calling application, the transaction is guaranteed to exist in the database. Write-ahead logging and automatic rollback and roll-forward of transactions during the recovery phase of SQL Server startup ensure durability.

Transaction Dependencies

In addition to supporting all four ACID properties, a transaction might exhibit several other behaviors. Some people call these behaviors "dependency problems" or "consistency problems," but I don't necessarily think of them as problems. They are merely possible behaviors, and except for lost updates, which are never considered desirable, you can determine which of these behaviors you want to allow and which you want to avoid. Your choice of isolation level determines which of these behaviors is allowed.

Lost Updates

Lost updates occur when two processes read the same data and both manipulate the data, changing its value, and then both try to update the original data to the new value. The second process might overwrite the first update completely. For example, suppose that two clerks in a receiving room are receiving parts and adding the new shipments to the inventory database. Clerk A and Clerk B both receive shipments of widgets. They both check the current inventory and see that 25 widgets are currently in stock. Clerk A's shipment has 50 widgets, so he adds 50 to 25 and updates the current value to 75. Clerk B's shipment has 20 widgets, so she adds 20 to the value of 25 that she originally read and updates the current value to 45, completely overriding the 50 new widgets that Clerk A processed. Clerk A's update is lost.

Lost updates are only one of the behaviors described here that you probably want to avoid in all cases.

Dirty Reads

Dirty reads occur when a process reads uncommitted data. If one process has changed data but not yet committed the change, another process reading the data will read it in an inconsistent state. For example, say that Clerk A has updated the old value of 25 widgets to 75, but before he commits, a salesperson looks at the current value of 75 and commits to sending 60 widgets to a customer the following day. If Clerk A then realizes that the widgets are defective and sends them back to the manufacturer, the salesperson has done a dirty read and taken action based on uncommitted data.

By default, dirty reads are not allowed. Keep in mind that the process updating the data has no control over whether another process can read its data before the first process is committed. It's up to the process reading the data to decide whether it wants to read data that is not guaranteed to be committed.

Nonrepeatable Reads

A read is nonrepeatable if a process might get different values when reading the same data in two separate reads within the same transaction. This can happen when another process changes the data in between the reads that the first process is doing. In the receiving room example, suppose that a manager comes in to do a spot check of the current inventory. She walks up to each clerk, asking the total number of widgets received today and adding the numbers on her calculator. When she's done, she wants to double-check the result, so she goes back to the first clerk. However, if Clerk A received more widgets between the manager's first and second inquiries, the total is different and the reads are nonrepeatable. Nonrepeatable reads are also called *inconsistent analysis*.

Phantoms

Phantoms occur when membership in a set changes. It can happen only when a query with a predicate—such as WHERE count_of_widgets < 10—is involved. A phantom occurs

if two *SELECT* operations using the same predicate in the same transaction return a different number of rows. For example, let's say that our manager is still doing spot checks of inventory. This time, she goes around the receiving room and notes which clerks have fewer than 10 widgets. After she completes the list, she goes back around to offer advice to everyone with a low total. However, if during her first walkthrough, a clerk with fewer than 10 widgets returned from a break but was not spotted by the manager, that clerk is not on the manager's list even though he meets the criteria in the predicate. This additional clerk (or row) is considered to be a phantom.

The behavior of your transactions depends on the isolation level. As mentioned earlier, you can decide which of the behaviors described previously to allow by setting an appropriate isolation level using the command *SET TRANSACTION ISOLATION LEVEL* <isolation_level>. Your concurrency model (optimistic or pessimistic) determines how the isolation level is implemented—or, more specifically, how SQL Server guarantees that the behaviors you don't want will not occur.

Isolation Levels

SQL Server 2008 supports five isolation levels that control the behavior of your read operations. Three of them are available only with pessimistic concurrency, one is available only with optimistic concurrency, and one is available with either. We look at these levels now, but a complete understanding of isolation levels also requires an understanding of locking and row versioning. In my descriptions of the isolation levels, I mention the locks or row versions that support that level, but keep in mind that locking and row versioning are discussed in detail later in the chapter.

Read Uncommitted

In Read Uncommitted isolation, all the behaviors described previously, except lost updates, are possible. Your queries can read uncommitted data, and both nonrepeatable reads and phantoms are possible. Read Uncommitted isolation is implemented by allowing your read operations to not take any locks, and because SQL Server isn't trying to acquire locks, it won't be blocked by conflicting locks acquired by other processes. Your process is able to read data that another process has modified but not yet committed.

In addition to reading individual values that are not yet committed, the Read Uncommitted isolation level introduces other undesirable behaviors. When using this isolation level and scanning an entire table, SQL Server can decide to do an allocation order scan (in page-number order), instead of a logical order scan (which would follow the page pointers). If there are concurrent operations by other processes that change data and move rows to a new location in the table, your allocation order scan can end up reading the same row twice. This can happen when you've read a row before it is updated, and then the update moves the row to a higher page number than your scan encounters later. In addition, performing an

allocation order scan under Read Uncommitted can cause you to miss a row completely. This can happen when a row on a high page number that hasn't been read yet is updated and moved to a lower page number that has already been read.

Although this scenario isn't usually the ideal option, with Read Uncommitted, you can't get stuck waiting for a lock, and your read operations don't acquire any locks that might affect other processes that are reading or writing data.

When using Read Uncommitted, you give up the assurance of strongly consistent data in favor of high concurrency in the system without users locking each other out. So when should you choose Read Uncommitted? Clearly, you don't want to use it for financial transactions in which every number must balance. But it might be fine for certain decision-support analyses—for example, when you look at sales trends—for which complete precision isn't necessary and the tradeoff in higher concurrency makes it worthwhile. Read Uncommitted isolation is a pessimistic solution to the problem of too much blocking activity because it just ignores the locks and does not provide you with transactional consistency.

Read Committed

SQL Server 2008 supports two varieties of Read Committed isolation, which is the default isolation level. This isolation level can be either optimistic or pessimistic, depending on the database setting READ_COMMITTED_SNAPSHOT. Because the default for the database option is off, the default for this isolation level is to use pessimistic concurrency control. Unless indicated otherwise, when I refer to the Read Committed isolation level, I am referring to both variations of this isolation level. I refer to the pessimistic implementation as Read Committed (locking), and I refer to the optimistic implementation as Read Committed (snapshot).

Read Committed isolation ensures that an operation never reads data that another application has changed but not yet committed. (That is, it never reads data that logically never existed.) With Read Committed (locking), if another transaction is updating data and consequently has exclusive locks on data rows, your transaction must wait for those locks to be released before you can use that data (whether you're reading or modifying). Also, your transaction must put share locks (at a minimum) on the data that are visited, which means that data might be unavailable to others to use. A share lock doesn't prevent others from reading the data, but it makes them wait to update the data. By default, share locks can be released after the data has been processed—they don't have to be held for the duration of the transaction, or even for the duration of the statement. (That is, if shared row locks are acquired, each row lock can be released as soon as the row is processed, even though the statement might need to process many more rows.)

Read Committed (snapshot) also ensures that an operation never reads uncommitted data, but not by forcing other processes to wait. In Read Committed (snapshot), every time a row is updated, SQL Server generates a version of the changed row with its previous committed values. The data being changed is still locked, but other processes can see the previous versions of the data as it was before the update operation began.

Repeatable Read

Repeatable Read is a pessimistic isolation level. It adds to the properties of Committed Read by ensuring that if a transaction revisits data or a query is reissued, the data does not change. In other words, issuing the same query twice within a transaction cannot pick up any changes to data values made by another user's transaction because no changes can be made by other transactions. However, the Repeatable Read isolation level does allow phantom rows to appear.

Preventing nonrepeatable reads is a desirable safeguard in some cases. But there's no free lunch. The cost of this extra safeguard is that all the shared locks in a transaction must be held until the completion (*COMMIT* or *ROLLBACK*) of the transaction. (Exclusive locks must always be held until the end of a transaction, no matter what the isolation level or concurrency model, so that a transaction can be rolled back if necessary. If the locks were released sooner, it might be impossible to undo the work because other concurrent transactions might have used the same data and changed the value.) No other user can modify the data visited by your transaction as long as your transaction is open. Obviously, this can seriously reduce concurrency and degrade performance. If transactions are not kept short or if applications are not written to be aware of such potential lock contention issues, SQL Server can appear to stop responding when a process is waiting for locks to be released.

Note You can control how long SQL Server waits for a lock to be released by using the session option LOCK_TIMEOUT. It is a SET option, so the behavior can be controlled only for an individual session. There is no way to set a LOCK_TIMEOUT value for SQL Server as a whole. You can read about LOCK_TIMEOUT in *SQL Server Books Online*.

Snapshot

Snapshot isolation (sometimes referred to as SI) is an optimistic isolation level. Like Read Committed (snapshot), it allows processes to read older versions of committed data if the current version is locked. The difference between Snapshot and Read Committed (snapshot) has to do with how old the older versions have to be. (We see the details in the section entitled "Row Versioning," later in this chapter.) Although the behaviors prevented by Snapshot isolation are the same as those prevented by Serializable, Snapshot is not truly a Serializable isolation level. With Snapshot isolation, it is possible to have two transactions executing simultaneously that give us a result that is not possible in any serial execution. Table 10-1 shows an example of two simultaneous transactions. If they run in parallel, they end up switching the price of two books in the *titles* table in the *pubs* database. However, there is no serial execution that would end up switching the values, whether we run Transaction 1 and then Transaction 2, or run Transaction 2 and then Transaction 1. Either serial order ends up with the two books having the same price.

TABLE 10-1 Two Simultaneous Transactions in Snapshot Isolation That Cannot Be Run Serially

Time	Transaction 1	Transaction 2
1	USE pubs SET TRANSACTION ISOLATION LEVEL SNAPSHOT DECLARE @price money BEGIN TRAN	USE pubs SET TRANSACTION ISOLATION LEVEL SNAPSHOT DECLARE @price money BEGIN TRAN
2	SELECT @price = price FROM titles WHERE title_id = 'BU1032'	SELECT @price = price FROM titles WHERE title_id = 'PS7777'
3	UPDATE titles SET price = @price WHERE title_id = 'PS7777'	UPDATE titles SET price = @price WHERE title_id = 'BU1032'
4	COMMIT TRAN	COMMIT TRAN

Serializable

Serializable is also a pessimistic isolation level. The Serializable isolation level adds to the properties of Repeatable Read by ensuring that if a query is reissued, rows were not added in the interim. In other words, phantoms do not appear if the same query is issued twice within a transaction. Serializable is therefore the strongest of the pessimistic isolation levels because it prevents all the possible undesirable behaviors discussed earlier—that is, it does not allow uncommitted reads, nonrepeatable reads, or phantoms, and it also guarantees that your transactions can be run serially.

Preventing phantoms is another desirable safeguard. But once again, there's no free lunch. The cost of this extra safeguard is similar to that of Repeatable Read—all the shared locks in a transaction must be held until the transaction completes. In addition, enforcing the Serializable isolation level requires that you not only lock data that has been read, but also lock data that does not exist! For example, suppose that within a transaction, we issue a *SELECT* statement to read all the customers whose ZIP code is between 98000 and 98100, and on first execution, no rows satisfy that condition. To enforce the Serializable isolation level, we must lock that range of potential rows with ZIP codes between 98000 and 98100 so that if the same query is reissued, there are still no rows that satisfy the condition. SQL Server handles this situation by using a special kind of lock called a *key-range lock*. Key-range locks require that there be an index on the column that defines the range of values. (In this example, that would be the column containing the ZIP codes.) If there is no index on that column, Serializable isolation requires a table lock. I discuss the different types of locks in detail in the section on locking. The Serializable level gets its name from the fact that running multiple serializable transactions at the same time is the equivalent of running them one at a time—that is, serially.

For example, suppose that transactions A, B, and C run simultaneously at the Serializable level and each tries to update the same range of data. If the order in which the transactions acquire locks on the range of data is B, C, and then A, the result obtained by running all three

simultaneously is the same as if they were run sequentially in the order B, C, and then A. Serializable does not imply that the order is known in advance. The order is considered a chance event. Even on a single-user system, the order of transactions hitting the queue would be essentially random. If the batch order is important to your application, you should implement it as a pure batch system. Serializable means only that there should be a way to run the transactions serially to get the same result you get when you run them simultaneously. Table 10-1 illustrates a case where two transactions cannot be run serially and get the same result.

Table 10-2 summarizes the behaviors that are possible in each isolation level and notes the concurrency control model that is used to implement each level. You can see that Read Committed and Read Committed (snapshot) are identical in the behaviors they allow, but the behaviors are implemented differently—one is pessimistic (locking), and one is optimistic (row versioning). Serializable and Snapshot also have the same No values for all the behaviors, but one is pessimistic and one is optimistic.

TABLE 10-2 Behaviors Allowed in Each Isolation Level

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom	Concurrency Control
Read Uncommitted	Yes	Yes	Yes	Pessimistic
Read Committed (locking)	No	Yes	Yes	Pessimistic
Read Committed (snapshot)	No	Yes	Yes	Optimistic
Repeatable Read	No	No	Yes	Pessimistic
Snapshot	No	No	No	Optimistic
Serializable	No	No	No	Pessimistic

Locking

Locking is a crucial function of any multiuser database system, including SQL Server. Locks are applied in both the pessimistic and optimistic concurrency models, although the way other processes deal with locked data is different in each. The reason I refer to the pessimistic variation of Read Committed isolation as Read Committed (locking) is because locking allows concurrent transactions to maintain consistency. In the pessimistic model, writers always block readers and writers, and readers can block writers. In the optimistic model, the only blocking that occurs is that writers block other writers. But to really understand what these simplified behavior summaries mean, we need to look at the details of SQL Server locking.

Locking Basics

SQL Server can lock data using several different modes. For example, read operations acquire shared locks, and write operations acquire exclusive locks. Update locks are acquired during the initial portion of an update operation, while SQL Server is searching for the data

to update. SQL Server acquires and releases all these types of locks automatically. It also manages compatibility between lock modes, resolves deadlocks, and escalates locks if necessary. It controls locks on tables, on the pages of a table, on index keys, and on individual rows of data. Locks can also be held on system data—data that's private to the database system, such as page headers and indexes.

SQL Server provides two separate locking systems. The first system affects all fully shared data and provides row locks, page locks, and table locks for tables, data pages, Large Object (LOB) pages, and leaf-level index pages. The second system is used internally for index concurrency control, controlling access to internal data structures and retrieving individual rows of data pages. This second system uses latches, which are less resource-intensive than locks and provide performance optimizations. You could use full-blown locks for all locking, but because of their complexity, they would slow down the system if you used them for all internal needs. If you examine locks using the *sp_lock* system stored procedure or a similar mechanism that gets information from the *sys.dm_tran_locks* view, you cannot see latches—you see only information about locks.

Another way to look at the difference between locks and latches is that locks ensure the logical consistency of the data and latches ensure the physical consistency. Latching happens when you place a row physically on a page or move data in other ways, such as compressing the space on a page. SQL Server must guarantee that this data movement can happen without interference.

Spinlocks

For shorter-term needs, SQL Server achieves mutual exclusion with a spinlock. Spinlocks are used purely for mutual exclusion and never to lock user data. They are even more lightweight than latches, which are lighter than the full locks used for data and index leaf pages. The requester of a spinlock repeats its request if the lock is not immediately available. (That is, the requester "spins" on the lock until it is free.)

Spinlocks are often used as mutexes within SQL Server for resources that are usually not busy. If a resource is busy, the duration of a spinlock is short enough that retrying is better than waiting and then being rescheduled by the operating system, which results in context switching between threads. The savings in context switches more than offsets the cost of spinning as long as you don't have to spin too long. Spinlocks are used for situations in which the wait for a resource is expected to be brief (or if no wait is expected). The *sys.dm_os_tasks* dynamic management view (DMV) shows a status of SPINLOOP for any task that is currently using a spinlock.

Lock Types for User Data

We examine four aspects of locking user data. First we look at the mode of locking (the type of lock). I already mentioned shared, exclusive, and update locks, and I go into more detail

about these modes as well as others. Next we look at the granularity of the lock, which specifies how much data is covered by a single lock. This can be a row, a page, an index key, a range of index keys, an extent, a partition, or an entire table. The third aspect of locking is the duration of the lock. As mentioned earlier, some locks are released as soon as the data has been accessed, and some locks are held until the transaction commits or rolls back. The fourth aspect of locking concerns the ownership of the lock (the scope of the lock). Locks can be owned by a session, a transaction, or a cursor.

Lock Modes

SQL Server uses several locking modes, including shared locks, exclusive locks, update locks, and intent locks, plus variations on these. It is the mode of the lock that determines whether a concurrently requested lock is compatible with locks that have already been granted. We see the lock compatibility matrix at the end of this section in Figure 10-2.

Shared Locks

Shared locks are acquired automatically by SQL Server when data is read. Shared locks can be held on a table, a page, an index key, or an individual row. Many processes can hold shared locks on the same data, but no process can acquire an exclusive lock on data that has a shared lock on it (unless the process requesting the exclusive lock is the same process as the one holding the shared lock). Normally, shared locks are released as soon as the data has been read, but you can change this by using query hints or a different transaction isolation level.

Exclusive Locks

SQL Server automatically acquires exclusive locks on data when the data is modified by an *INSERT, UPDATE,* or *DELETE* operation. Only one process at a time can hold an exclusive lock on a particular data resource; in fact, as you see when we discuss lock compatibility later in this chapter, no locks of any kind can be acquired by a process if another process has the requested data resource exclusively locked. Exclusive locks are held until the end of the transaction. This means the changed data is normally not available to any other process until the current transaction commits or rolls back. Other processes can decide to read exclusively locked data by using query hints.

Update Locks

Update locks are really not a separate kind of lock; they are a hybrid of shared and exclusive locks. They are acquired when SQL Server executes a data modification operation but first, SQL Server needs to search the table to find the resource that needs to be modified. Using query hints, a process can specifically request update locks, and in that case, the update locks prevent the conversion deadlock situation presented in Figure 10-6 later in this chapter.

Update locks provide compatibility with other current readers of data, allowing the process to later modify data with the assurance that the data hasn't been changed since it was last read. An update lock is not sufficient to allow you to change the data—all modifications require that the data resource being modified have an exclusive lock. An update lock acts as a serialization gate to queue future requests for the exclusive lock. (Many processes can hold shared locks for a resource, but only one process can hold an update lock.) So long as a process holds an update lock on a resource, no other process can acquire an update lock or an exclusive lock for that resource; instead, another process requesting an update or exclusive lock for the same resource must wait. The process holding the update lock can convert it into an exclusive lock on that resource because the update lock prevents lock incompatibility with any other processes. You can think of update locks as "intent-to-update" locks, which is essentially the role they perform. Used alone, update locks are insufficient for updating data—an exclusive lock is still required for actual data modification. Serializing access for the exclusive lock lets you avoid conversion deadlocks. Update locks are held until the end of the transaction or until they are converted to an exclusive lock.

Don't let the name fool you: update locks are not just for *UPDATE* operations. SQL Server uses update locks for any data modification operation that requires a search for the data prior to the actual modification. Such operations include qualified updates and deletes, as well as inserts into a table with a clustered index. In the latter case, SQL Server must first search the data (using the clustered index) to find the correct position at which to insert the new row. While SQL Server is only searching, it uses update locks to protect the data; only after it has found the correct location and begins inserting does it convert the update lock to an exclusive lock.

Intent Locks

Intent locks are not really a separate mode of locking; they are a qualifier to the modes previously discussed. In other words, you can have intent shared locks, intent exclusive locks, and even intent update locks. Because SQL Server can acquire locks at different levels of granularity, a mechanism is needed to indicate that a component of a resource is already locked. For example, if one process tries to lock a table, SQL Server needs a way to determine whether a row (or a page) of that table is already locked. Intent locks serve this purpose. We discuss them in more detail when we look at lock granularity.

Special Lock Modes

SQL Server offers three additional lock modes: schema stability locks, schema modification locks, and bulk update locks. When queries are compiled, schema stability locks prevent other processes from acquiring schema modification locks, which are taken when a table's structure is being modified. A bulk update lock is acquired when the *BULK INSERT* command is executed or when the bcp utility is run to load data into a table. In addition, the bulk import operation must request this special lock by using the TABLOCK hint. Alternatively, the table owner can set the table option called *table lock on bulk load* to True, and then any bulk copy *IN* or *BULK INSERT* operation automatically requests a bulk update lock. Requesting

this special bulk update table lock does not necessarily mean it is granted. If other processes already hold locks on the table, or if the table has any indexes, a bulk update lock cannot be granted. If multiple connections have requested and received a bulk update lock, they can perform parallel loads into the same table. Unlike exclusive locks, bulk update locks do not conflict with each other, so concurrent inserts by multiple connections is supported.

Conversion Locks

Conversion locks are never requested directly by SQL Server, but are the result of a conversion from one mode to another. The three types of conversion locks supported by SQL Server 2008 are SIX, SIU, and UIX. The most common of these is the SIX, which occurs if a transaction is holding a shared (S) lock on a resource and later an IX lock is needed. The lock mode is indicated as SIX. For example, suppose that you issue the following batch:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRAN
SELECT * FROM bigtable
UPDATE bigtable
    SET col = 0
    WHERE keycolumn = 100
```

If the table is large, the *SELECT* statement acquires a shared table lock. (If the table has only a few rows, SQL Server acquires individual row or key locks.) The *UPDATE* statement then acquires a single exclusive key lock to perform the update of a single row, and the X lock at the key level means an IX lock at the page and table level. The table then shows SIX when viewed through *sys.dm_tran_locks*. Similarly, SIU occurs when a process has a shared lock on a table and an update lock on a row of that table, and UIX occurs when a process has an update lock on the table and an exclusive lock on a row.

Table 10-3 shows most of the lock modes, as well as the abbreviations used in *sys.dm_tran_locks*.

TABLE 10-3 SQL Server Lock Modes

Abbreviation	Lock Mode	Description
S	Shared	Allows other processes to read but not change the locked resource.
X	Exclusive	Prevents another process from modifying or reading data in the locked resource.
U	Update	Prevents other processes from acquiring an update or exclusive lock; used when searching for the data to modify.
IS	Intent shared	Indicates that a component of this resource is locked with a shared lock. This lock can be acquired only at the table or page level.
IU	Intent update	Indicates that a component of this resource is locked with an update lock. This lock can be acquired only at the table or page level.

TABLE 10-3 SQL Server Lock Modes

Abbreviation	Lock Mode	Description
IX	Intent exclusive	Indicates that a component of this resource is locked with an exclusive lock. This lock can be acquired only at the table or page level.
SIX	Shared with intent exclusive	Indicates that a resource holding a shared lock also has a component (a page or row) locked with an exclusive lock.
SIU	Shared with intent update	Indicates that a resource holding a shared lock also has a component (a page or row) locked with an update lock.
UIX	Update with intent exclusive	Indicates that a resource holding an update lock also has a component (a page or row) locked with an exclusive lock.
Sch-S	Schema stability	Indicates that a query using this table is being compiled.
Sch-M	Schema modification	Indicates that the structure of the table is being changed.
BU	Bulk update	Used when a bulk copy operation is copying data into a table and the TABLOCK hint is being applied (either manually or automatically).

Key-Range Locks

Additional lock modes—called *key-range locks*—are taken only in the Serializable isolation level for locking ranges of data. Most lock modes can apply to almost any lock resource. For example, shared and exclusive locks can be taken on a table, a page, a row, or a key. Because key-range locks can be taken only on keys, I describe the details of key-range locks later in this chapter in the section on key locks.

Lock Granularity

SQL Server can lock user data resources (not system resources, which are protected with latches) at the table, page, or row level. (If locks are escalated, SQL Server can also lock a single partition of a table or index.) In addition, SQL Server can lock index keys and ranges of index keys. Figure 10-1 shows the basic lock levels in a table that can be acquired when a resource is first accessed. Keep in mind that if the table has a clustered index, the data rows are at the leaf level of the clustered index and they are locked with key locks instead of row locks.

The *sys.dm_tran_locks* view keeps track of each lock and contains information about the resource, which is locked (such as a row, key, or page), the mode of the lock, and an identifier for the specific resource. Keep in mind that *sys.dm_tran_locks* is only a dynamic view that is used to display the information about the locks that are held. The actual information is stored in internal SQL Server structures that are not visible to us at all. So when I talk about information being in the *sys.dm_tran_locks* view, I am referring to the fact that the information can be seen through that view.

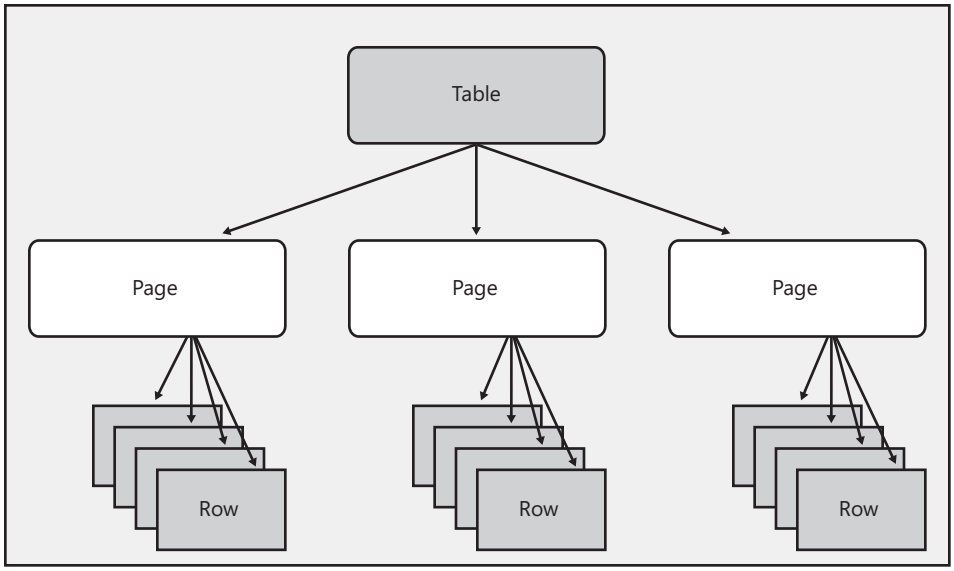


FIGURE 10-1 Levels of granularity for SQL Server locks on a table

When a process requests a lock, SQL Server compares the lock requested to the resources already listed in *sys.dm_tran_locks* and looks for an exact match on the resource type and identifier. However, if one process has a row exclusively locked in the *Sales.SalesOrderHeader* table, for example, another process might try to get a lock on the entire *Sales.SalesOrderHeader* table. Because these are two different resources, SQL Server does not find an exact match unless additional information is already in *sys.dm_tran_locks*. This is what intent locks are for. The process that has the exclusive lock on a row of the *Sales.SalesOrderHeader* table also has an intent exclusive lock on the page containing the row and another intent exclusive lock on the table containing the row. We can see those locks by first running this code:

```
USE Adventureworks2008;
BEGIN TRAN
UPDATE  Sales.SalesOrderHeader
SET ShipDate = ShipDate + 1
WHERE SalesOrderID = 43666;
```

This statement should affect a single row. Because I have started a transaction and not yet terminated it, the exclusive locks acquired are still held. I can look at those locks using the *sys.dm_tran_locks view:*

```
SELECT resource_type, resource_description,
        resource_associated_entity_id, request_mode, request_status
FROM sys.dm_tran_locks
WHERE resource_associated_entity_id > 0;
```

I give you more details about the data in the section entitled "*sys.dm_tran_locks*" later in this chapter, but for now, just note that the reason for the filter in the WHERE clause is that I am

interested only in locks that are actually held on data resources. If you are running a query on a SQL Server instance that others are using, you might have to provide more filters to get just the rows you're interested in. For example, you could include a filter on *request_session_id* to limit the output to locks held by a particular session. Your results should look something like this:

```
resource_type resource_description resource_associated_entity_id request_mode request_status
------------- -------------------- ------------------------------ ------------ ---------------
KEY           (92007ad11d1d)       72057594045857792              X            GRANT
PAGE          1:5280               72057594045857792              IX           GRANT
OBJECT                             722101613                      IX           GRANT
```

Note that there are three locks, even though the *UPDATE* statement affected only a single row. For the KEY and the PAGE locks, the *resource_associated_entity_id* is a partition_id. For the OBJECT locks, the *resource_associated_entity_id* is a table. We can verify what table it is by using the following query:

```
SELECT object_name(722101613)
```

The results should tell us that the object is the *Sales.SalesOrderHeader* table. When the second process attempts to acquire an exclusive lock on that table, it finds a conflicting row already in *sys.dm_tran_locks* on the same lock resource (the *Sales.SalesOrderHeader* table), and it is blocked. The *sys.dm_tran_locks* view shows us the following row, indicating a request for an exclusive lock on an object that is unable to be granted. The process requesting the lock is in a WAIT state:

```
resource_type resource_description resource_associated_entity_id  request_mode request_status
------------- -------------------- ------------------------------ ------------ ------------
OBJECT                             722101613                      X            WAIT
```

Not all requests for locks on resources that are already locked result in a conflict. A conflict occurs when one process requests a lock on a resource that is already locked by another process in an incompatible lock mode. For example, two processes can each acquire shared locks on the same resource because shared locks are compatible with each other. I discuss lock compatibility in detail later in this chapter.

Key Locks

SQL Server 2008 supports two kinds of key locks, and which one it uses depends on the isolation level of the current transaction. If the isolation level is Read Committed, Repeatable Read, or Snapshot, SQL Server tries to lock the actual index keys accessed while processing the query. With a table that has a clustered index, the data rows are the leaf level of the index, and you see key locks acquired. If the table is a heap, you might see key locks for the nonclustered indexes and row locks for the actual data.

If the isolation level is Serializable, the situation is different. We want to prevent phantoms, so if we have scanned a range of data within a transaction, we need to lock enough of the table

to make sure no one can insert a value into the range that was scanned. For example, we can issue the following query within an explicit transaction in the *AdventureWorks2008* database:

```
BEGIN TRAN
SELECT * FROM Sales.SalesOrderHeader
WHERE CustomerID BETWEEN 100 and 110;
```

When you use Serializable isolation, locks must be acquired to make sure no new rows with *CustomerID* values between 100 and 110 are inserted before the end of the transaction. Much older versions of SQL Server (prior to 7.0) guaranteed this by locking whole pages or even the entire table. In many cases, however, this was too restrictive—more data was locked than the actual WHERE clause indicated, resulting in unnecessary contention. SQL Server 2008 uses the key-range locks mode, which is associated with a particular key value in an index and indicates that all values between that key and the previous one in the index are locked.

The *AdventureWorks2008* database includes an index on the *Person* table with the *LastName* column as the leading column. Assume that we are in TRANSACTION ISOLATION LEVEL SERIALIZABLE and we issue this *SELECT* statement inside a user-defined transaction:

```
SELECT * FROM Person.Person
WHERE LastName BETWEEN 'Freller' AND 'Freund';
```

If *Fredericksen, French,* and *Friedland* are sequential leaf-level index keys in an index on the *LastName* column, the second two of these keys (*French* and *Friedland*) acquire key-range locks (although only one row, for *French,* is returned in the result set). The key-range locks prevent any inserts into the ranges ending with the two key-range locks. No values greater than *Fredericksen* and less than or equal to *French* can be inserted, and no values greater than *French* and less than or equal to *Friedland* can be inserted. Note that the key-range locks imply an open interval starting at the previous sequential key and a closed interval ending at the key on which the lock is placed. These two key-range locks prevent anyone from inserting either *Fremlich* or *Frenkin,* which are in the range specified in the WHERE clause. However, the key-range locks would also prevent anyone from inserting *Freedman* (which is greater than *Fredericksen* and less than *French*), even though *Freedman* is not in the query's specified range. Key-range locks are not perfect, but they do provide much greater concurrency than locking whole pages or tables, while guaranteeing that phantoms are prevented.

There are nine types of key-range locks, and each has a two-part name: the first part indicates the type of lock on the range of data between adjacent index keys, and the second part indicates the type of lock on the key itself. These nine types of key-range locks are described in Table 10-4.

TABLE 10-4 Types of Key-Range Locks

Abbreviation	Description
RangeS-S	Shared lock on the range between keys; shared lock on the key at the end of the range
RangeS-U	Shared lock on the range between keys; update lock on the key at the end of the range
RangeIn-Null	Exclusive lock to prevent inserts on the range between keys; no lock on the keys themselves
RangeX-X	Exclusive lock on the range between keys; exclusive lock on the key at the end of the range
RangeIn-S	Conversion lock created by S and RangeIn_Null lock
RangeIn-U	Conversion lock created by U and RangeIn_Null lock
RangeIn-X	Conversion of X and RangeIn_Null lock
RangeX-S	Conversion of RangeIn_Null and RangeS_S lock
RangeX-U	Conversion of RangeIn_Null and RangeS_U lock

Many of these lock modes are very rare or transient, so you do not see them very often in *sys.dm_tran_locks*. For example, the RangeIn-Null lock is acquired when SQL Server attempts to insert into the range between keys in a session using Serializable isolation. This type of lock is not often seen because it is typically very transient. It is held only until the correct location for insertion is found, and then the lock is converted into an X lock. However, if one transaction scans a range of data using the Serializable isolation level and then another transaction tries to insert into that range, the second transaction has a lock request with a WAIT status with the RangeIn-Null mode. You can observe this by looking at the status column in *sys.dm_tran_locks*, which we discuss in more detail later in the chapter.

Additional Lock Resources

In addition to locks on objects, pages, keys, and rows, a few other resources can be locked by SQL Server. Locks can be taken on extents—units of disk space that are 64 KB in size (eight pages of 8 KB each). This kind of locking occurs automatically when a table or an index needs to grow and a new extent must be allocated. You can think of an extent lock as another type of special-purpose latch, but it does show up in *sys.dm_tran_locks*. Extents can have both shared extent and exclusive extent locks.

When you examine the contents of *sys.dm_tran_locks*, you should notice that most processes hold a lock on at least one database (*resource_type* = DATABASE). In fact, any process holding locks in any database other than *master* or *tempdb* has a lock for that database resource. These database locks are always shared locks if the process is just using the database. SQL Server checks for these database locks when determining whether a database is in use, and then it can determine whether the database can be dropped, restored, altered, or closed. Because few changes can be made to *master* and *tempdb* and they cannot be dropped

or closed, DATABASE locks are unnecessary. In addition, *tempdb* is never restored, and to restore the *master* database, the entire server must be started in single-user mode, so again, DATABASE locks are unnecessary. When attempting to perform one of these operations, SQL Server requests an exclusive database lock, and if any other processes have a shared lock on the database, the request blocks. Generally, you don't need to be concerned with extent or database locks, but you see them if you are perusing *sys.dm_tran_locks*.

You might occasionally see locks on ALLOCATION_UNIT resources. Although all table and index structures contain one or more ALLOCATION_UNITs, when these locks occur, it means SQL Server is dealing with one of these resources that is no longer tied to a particular object. For example, when you drop or rebuild large tables or indexes, the actual page deallocation is deferred until after the transaction commits. Deferred drop operations do not release allocated space immediately, and they introduce additional overhead costs, so a deferred drop is done only on tables or indexes that use more than 128 extents. If the table or index uses 128 or fewer extents, dropping, truncating, and rebuilding are not deferred operations. During the first phase of a deferred operation, the existing allocation units used by the table or index are marked for deallocation and locked until the transaction commits. This is where you see ALLOCATION_UNIT locks in *sys.dm_tran_locks*. You can also look in the *sys.allocation_units* view to find allocation units with a *type_desc* value of DROPPED to see how much space is being used by the allocation units that are not available for reuse but are not currently part of any object. The actual physical dropping of the allocation unit's space occurs after the transaction commits.

Finally, you occasionally have locks on individual partitions, which are indicated in the lock metadata as HOBT locks. This can happen only when locks are escalated, and only if you have specified that escalation to the partition level is allowed (and, of course, only when the table or index has been partitioned). We look at how you can specify that you want partition-level locking in the section entitled "Lock Escalation," later in this chapter.

Identifying Lock Resources

When SQL Server tries to determine whether a requested lock can be granted, it checks the *sys.dm_tran_locks* view to determine whether a matching lock with a conflicting lock mode already exists. It compares locks by looking at the database ID (*resource_database_ID),* the values in the *resource_description* and *resource_associated_entity_id* columns, and the type of resource locked. SQL Server knows nothing about the meaning of the resource description. It simply compares the strings identifying the lock resources to look for a match. If it finds a match with a *request_status* value of GRANT, it knows the resource is already locked; it then uses the lock compatibility matrix to determine whether the current lock is compatible with the one being requested. Table 10-5 shows many of the possible lock resources that are displayed in the first column of the *sys.dm_tran_locks* view and the information in the *resource_description* column, which is used to define the actual resource locked.

TABLE 10-5 Lockable Resources in SQL Server

Resource_Type	Resource_Description	Example
DATABASE	None; the database is always indicated in the *resource_database_ID* column for every locked resource.	12
OBJECT	The object ID (which can be any database object, not necessarily a table) is reported in the *resource_associated_entity_id* column.	69575286
HOBT	*hobt_id* is reported in the *resource_associated_entity_id* column. Used only when partition locking has been enabled for a table.	72057594038779904
EXTENT	File number:page number of the first page of the extent.	1:96
PAGE	File number:page number of the actual table or index page.	1:104
KEY	A hashed value derived from all the key components and the locator. For a nonclustered index on a heap, where columns *c1* and *c2* are indexed, the hash will contain contributions from *c1, c2,* and the *RID*.	ac0001a10a00
ROW	File number:page number:slot number of the actual row.	1:161:3

Note that key locks and key-range locks have identical resource descriptions because key range is considered a mode of locking, not a locking resource. When you look at output from the *sys.dm_tran_locks* view, you see that you can distinguish between these types of locks by the value in the lock mode column.

Another type of lockable resource is METADATA. More than any other resource, METADATA resources are divided into multiple subtypes, which are described in the *resource_subtype* column of *sys.dm_tran_locks*. You might see dozens of subtypes of METADATA resources, but most of them are beyond the scope of this book. For some, however, even though *SQL Server Books Online* describes them as "for internal use only," it is pretty obvious what they refer to. For example, when you change properties of a database, you can see a *resource_type* of METADATA and a *resource_subtype* of DATABASE. The value in the *resource_description* column of that row is *database_id =<ID>*, indicating the ID of the database whose metadata is currently locked.

Associated Entity ID

For locked resources that are part of a larger entity, the *resource_associated_entity_id* column in *sys.dm_tran_locks* displays the ID of that associated entity in the database. This can be an object ID, a partition ID, or an allocation unit ID, depending on the resource type. Of course, for some resources, such as DATABASE and EXTENT, there is no *resource_associated_entity_id*. An *object ID* value is given in this column for OBJECT resources, and an allocation unit ID is given for ALLOCATION_UNIT resources. A partition ID is provided for resource types PAGE, KEY, and RID.

There is no simple function to convert a partition ID value to an object name; you have to actually select from the *sys.partitions* view. The following query translates all the *resource_associated_entity_id* values for locks in the current database by joining *sys.dm_tran_locks* to *sys.partitions*. For OBJECT resources, the *object_name* function is applied to the *resource_associated_entity_id* column. For PAGE, KEY, and RID resources, I use the *object_name* function with the *object_id* value from the *sys.partitions* view. For other resources for which there is no *resource_associated_entity_id*, the code just returns `n/a`. Because the code references the *sys.partitions* view, which occurs in each database, this code is filtered to return only lock information for resources in the current database. The output is organized to reflect the information returned by the *sp_lock* procedure, but you can add any additional filters or columns that you need. I will use this query in many examples later in this chapter, so I create a VIEW based on the *SELECT* and call it *DBlocks:*

```
CREATE VIEW DBlocks AS
SELECT request_session_id as spid,
    db_name(resource_database_id) as dbname,
    CASE
   WHEN resource_type = 'OBJECT' THEN
         object_name(resource_associated_entity_id)
      WHEN resource_associated_entity_id = 0 THEN 'n/a'
   ELSE object_name(p.object_id)
    END as entity_name, index_id,
       resource_type as resource,
       resource_description as description,
       request_mode as mode, request_status as status
FROM sys.dm_tran_locks t LEFT JOIN sys.partitions p
   ON p.partition_id = t.resource_associated_entity_id
WHERE resource_database_id = db_id();
```

Lock Duration

The length of time that a lock is held depends primarily on the mode of the lock and the transaction isolation level in effect. The default isolation level for SQL Server is Read Committed. At this level, shared locks are released as soon as SQL Server has read and processed the locked data. In Snapshot isolation, the behavior is the same—shared locks are released as soon as SQL Server has read the data. If your transaction isolation level is Repeatable Read or Serializable, shared locks have the same duration as exclusive locks; that is, they are not released until the transaction is over. In any isolation level, an exclusive lock is held until the end of the transaction, whether the transaction is committed or rolled back. An update lock is also held until the end of the transaction unless it has been promoted to an exclusive lock, in which case the exclusive lock, as is always the case with exclusive locks, remains for the duration of the transaction.

In addition to changing your transaction isolation level, you can control the lock duration by using query hints. I discuss query hints for locking, briefly, later in this chapter.

Lock Ownership

Lock duration is also directly affected by the lock ownership. Lock ownership has nothing to do with the process that requested the lock, but you can think of it as the "scope" of the lock. There are four types of lock owners, or lock scopes: transactions, cursors, transaction_workspaces, and sessions. The lock owner can be viewed through the *request_owner_type* column in the *sys.dm_tran_locks* view.

Most of our locking discussion deals with locks with a lock owner of TRANSACTION. As we've seen, these locks can have two different durations depending on the isolation level and lock mode. The duration of shared locks in Read Committed isolation is only as long as the locked data is being read. The duration of all other locks owned by a transaction is until the end of the transaction.

A lock with a *request_ownertype* value of CURSOR must be requested explicitly when the cursor is declared. If a cursor is opened using a locking mode of SCROLL_LOCKS, a cursor lock is held on every row fetched until the next row is fetched or the cursor is closed. Even if the transaction commits before the next fetch, the cursor lock is not released.

In SQL Server 2008, locks owned by a session must also be requested explicitly and apply only to APPLICATION locks. A session lock is requested using the *sp_getapplock* procedure. Its duration is until the session disconnects or the lock is released explicitly.

Transaction_workspace locks are acquired every time a database is accessed, and the resource associated with these locks is always a database. A workspace holds database locks for sessions that are enlisted into a common environment. Usually, there is one workspace per session, so all DATABASE locks acquired in the session are kept in the same workspace object. In the case of distributed transactions, multiple sessions are enlisted into the same workspace, so they share the database locks.

Every process acquires a DATABASE lock with an owner of SHARED_TRANSACTION_WORKSPACE on any database when the process issues the *USE* command. The exception is any processes that use *master* or *tempdb*, in which case no DATABASE lock is taken. That lock isn't released until another *USE* command is issued or until the process is disconnected. If a process attempts to *ALTER*, *RESTORE,* or *DROP* the database, the DATABASE lock acquired has an owner of EXCLUSIVE_TRANSACTION_WORKSPACE. SHARED_TRANSACTION_WORKSPACE and EXCLUSIVE_TRANSACTION_WORKSPACE locks are maintained by the same workspace and are just two different lists in one workspace. The use of two different owner names is misleading in this case.

Viewing Locks

To see the locks currently outstanding in the system, as well as those that are being waited for, the best source of information is the *sys.dm_tran_locks* view. I've shown you some queries

from this view in previous sections, and in this section, I show you a few more and explain what more of the output columns mean. This view replaces the *sp_lock* procedure. Although calling a procedure might require less typing than querying the *sys.dm_tran_locks* view, the view is much more flexible. Not only are there many more columns of information providing details about your locks, but as a view, *sys.dm_tran_locks* can be queried to select just the columns you want, or only the rows that meet your criteria. It can be joined with other views and aggregated to get summary information about how many locks of each kind are being held.

sys.dm_tran_locks

All the columns (with the exception of the last column called *lock_owner_address*) in *sys.dm_tran_locks* start with one of two prefixes. The columns whose names begin with *resource_* describe the resource on which the lock request is being made. The columns whose names begin with *request_* describe the process requesting the lock. Two requests operate on the same resource only if all the *resource_* columns are the same.

***resource_* Columns** I've mentioned most of the *resource_* columns already, but I referred only briefly to the *resource_subtype* column. Not all resources have subtypes, and some have many. The METADATA resource type, for example, has over 40 subtypes.

Table 10-6 lists all the subtypes for resource types other than METADATA.

TABLE 10-6 Subtype Resources

Resource Type	Resource Subtypes	Description
DATABASE	BULKOP_BACKUP_DB	Used for synchronization of database backups with bulk operations
	BULKOP_BACKUP_LOG	Used for synchronization of database log backups with bulk operations
	DDL	Used to synchronize Data Definition Language (DDL) operations with File Group operations (such as *DROP*)
	STARTUP	Used for database startup synchronization
TABLE	UPDSTATS	Used for synchronization of statistics updates on a table
	COMPILE	Used for synchronization of stored procedure compiles
	INDEX_OPERATION	Used for synchronization of index operations
HOBT	INDEX_REORGANIZE	Used for synchronization of heap or index reorganization operations
	BULK_OPERATION	Used for heap-optimized bulk load operations with concurrent scan, in the Snapshot, Read Uncommitted, and Read Committed SI levels
ALLOCATION_UNIT	PAGE_COUNT	Used for synchronization of allocation unit page count statistics during deferred drop operations

As previously mentioned, most METADATA subtypes are documented as being for INTERNAL USE ONLY, but their meaning is often pretty obvious. Each type of metadata can be locked separately as changes are made. Here is a partial list of the METADATA subtypes:

- INDEXSTATS
- STATS
- SCHEMA
- DATABASE_PRINCIPAL
- DB_PRINCIPAL_SID
- USER_TYPE
- DATA_SPACE
- PARTITION_FUNCTION
- DATABASE
- SERVER_PRINCIPAL
- SERVER

Most of the other METADATA subtypes not listed here refer to elements of SQL Server 2008 that are not discussed in this book, including CLR routines, XML, certificates, full-text search, and notification services.

***request_* Columns** I've also mentioned a couple of the most important *request_* columns in *sys.dm_tran_locks*, including *request_mode* (the type of lock requested), *request_owner_type* (the scope of the lock requested), and *request_session_id*. Here are some of the others:

- ***request_type*** In SQL Server 2008, the only type of resource request tracked in *sys.dm_tran_locks* is for a LOCK. Future versions may include other types of resources that can be requested.
- ***request_status*** Status can be one of three values: GRANT, CONVERT, or WAIT. A status of CONVERT indicates that the requestor has already been granted a request for the same resource in a different mode and is currently waiting for an upgrade (convert) from the current lock mode to be granted. (For example, SQL Server can convert a U lock to X.) A status of WAIT indicates that the requestor does not currently hold a granted request on the resource.
- ***request_reference_count*** This value is a rough count of number of times the same requestor has requested this resource and applies only to resources that are not automatically released at the end of a transaction. A granted resource is no longer considered to be held by a requestor if this field decreases to 0 and *request_lifetime* is also 0.
- ***request_lifetime*** This value is a code that indicates when the lock on the resource is released.

- ***request_session_id*** This value is the ID of the session that has requested the lock. The owning session ID can change for distributed and bound transactions. A value of –2 indicates that the request belongs to an orphaned DTC transaction. A value of –3 indicates that the request belongs to a deferred recovery transaction. (These are transactions whose rollback has been deferred at recovery because the rollback could not be completed successfully.)
- ***request_exec_context_id*** This value is the execution context ID of the process that currently owns this request. A value greater than 0 indicates that this is a subthread used to execute a parallel query.
- ***request_request_id*** This value is the request ID (batch ID) of the process that currently owns this request. This column is populated only for the requests coming in from a client application using Multiple Active Result Sets (MARS).
- ***request_owner_id*** This value is currently used only for requests with an owner of TRANSACTION, and the owner ID is the transaction ID. This column can be joined with the *transaction_id* column in the *sys.dm_tran_active_transactions* view.
- ***request_owner_guid*** This value is currently used only by DTC transactions when it corresponds to the DTC GUID for that transaction.
- ***lock_owner_address*** This value is the memory address of the internal data structure that is used to track this request. This column can be joined with the *resource_address* column in *sys.dm_os_waiting_tasks* if this request is in the WAIT or CONVERT state.

Locking Examples

The following examples show what many of the lock types and modes discussed earlier look like when reported using the *DBlocks* view that I described previously.

Example 1: *SELECT* with Default Isolation Level

SQL BATCH

```
USE Adventureworks2008;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN TRAN
SELECT * FROM Production.Product
WHERE Name = 'Reflector';
SELECT * FROM DBlocks WHERE spid = @@spid;
COMMIT TRAN
```

RESULTS FROM *DBlocks*

```
spid     dbname              entity_name  index_id  resource    description  mode  status
-----    ------------------  ------------ --------  ----------  -----------  ----- ------
60       Adventureworks2008  n/a          NULL      DATABASE                 S     GRANT
60       AdventureWorks2008  DBlocks      NULL      OBJECT                   IS    GRANT
```

There are no locks on the data in the *Production.Product* table because the batch was performing only *SELECT* operations that acquired shared locks. By default, the shared locks

are released as soon as the data has been read, so by the time the *SELECT* from the view is executed, the locks are no longer held. There is only the ever-present DATABASE lock, and an OBJECT lock on the view.

Example 2: *SELECT* with Repeatable Read Isolation Level

SQL BATCH

```
USE AdventureWorks2008;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRAN
SELECT * FROM Production.Product
WHERE Name LIKE 'Racing Socks%';
SELECT * FROM DBlocks
WHERE spid = @@spid
AND entity_name = 'Product';
COMMIT TRAN
```

RESULTS FROM *DBlocks*

```
spid dbname              entity_name  index_id   resource   description     mode     status
---- ------------------- ------------ ---------  ---------- --------------  -------- ------
54   AdventureWorks2008  Product      NULL       OBJECT                     IS       GRANT
54   AdventureWorks2008  Product      1          PAGE       1:16897         IS       GRANT
54   AdventureWorks2008  Product      1          KEY        (6b00b8eeda30)  S        GRANT
54   AdventureWorks2008  Product      1          KEY        (6a00dd896688)  S        GRANT
54   AdventureWorks2008  Product      3          KEY        (9502d56a217e)  S        GRANT
54   AdventureWorks2008  Product      3          PAGE       1:1767          IS       GRANT
54   AdventureWorks2008  Product      3          KEY        (9602945b3a67)  S        GRANT
```

This time, I filtered out the database lock and the locks on the view and the rowset, just to keep the focus on the data locks. Because the *Production.Product* table has a clustered index, the rows of data are all index rows in the leaf level. The locks on the two individual data rows returned are listed as key locks. There are also two key locks at the leaf level of the nonclustered index on the table used to find the relevant rows. In the *Production.Product* table, that nonclustered index is on the *Name* column. You can tell the clustered and nonclustered indexes apart by the value in the *index_id* column: the data rows (the leaf rows of the clustered index) have an *index_id* value of 1, and the nonclustered index rows have an *index_id* value of 3. (For nonclustered indexes, the *index_id* value can be anything between 2 and 250 or between 356 and 1005.) Because the transaction isolation level is Repeatable Read, the shared locks are held until the transaction is finished. Note that the index rows have shared (S) locks, and the data and index pages, as well as the table itself, have intent shared (IS) locks.

Example 3: *SELECT* with Serializable Isolation Level

SQL BATCH

```
USE AdventureWorks2008 ;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRAN
SELECT * FROM Production.Product
WHERE Name LIKE 'Racing Socks%';
```

```
SELECT * FROM DBlocks
WHERE spid = @@spid
AND entity_name = 'Product';
COMMIT TRAN
```

RESULTS FROM *DBlocks*

```
spid dbname              entity_name  index_id   resource   description    mode      status
---- ------------------ ------------ ---------- ---------- ------------   -------   ------
54   AdventureWorks2008 Product      NULL       OBJECT                    IS        GRANT
54   AdventureWorks2008 Product      1          PAGE       1:16897        IS        GRANT
54   AdventureWorks2008 Product      1          KEY        (6b00b8eeda30) S         GRANT
54   AdventureWorks2008 Product      1          KEY        (6a00dd896688) S         GRANT
54   AdventureWorks2008 Product      3          KEY        (9502d56a217e) RangeS-S  GRANT
54   AdventureWorks2008 Product      3          PAGE       1:1767         IS        GRANT
54   AdventureWorks2008 Product      3          KEY        (23027a50f6db) RangeS-S  GRANT
54   AdventureWorks2008 Product      3          KEY        (9602945b3a67) RangeS-S  GRANT
```

The locks held with the Serializable isolation level are almost identical to those held with the Repeatable Read isolation level. The main difference is in the mode of the lock. The two-part mode RangeS-S indicates a key-range lock in addition to the lock on the key itself. The first part (RangeS) is the lock on the range of keys between (and including) the key holding the lock and the previous key in the index. The key-range locks prevent other transactions from inserting new rows into the table that meet the condition of this query; that is, no new rows with a product name starting with *Racing Socks* can be inserted. The key-range locks are held on ranges in the nonclustered index on *Name* (*index_id* = 3) because that is the index used to find the qualifying rows. There are three key locks in the nonclustered index because three different ranges need to be locked. The two *Racing Socks* rows are *Racing Socks, L* and *Racing Socks, M*. SQL Server must lock the range from the key preceding the first *Racing Socks* row in the index up to the first *Racing Socks*. It must lock the range between the two rows starting with *Racing Socks,* and it must lock the range from the second *Racing Socks* to the next key in the index. (So actually nothing could be inserted between *Racing Socks* and the previous key, *Pinch Bolt,* or between *Racing Socks* and the next key, *Rear Brakes*. For example, we could not insert a product with the name *Portkey* or *Racing Tights*.)

Example 4: Update Operations

SQL BATCH

```
USE AdventureWorks2008;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN TRAN
UPDATE Production.Product
SET ListPrice = ListPrice * 0.6
WHERE Name LIKE 'Racing Socks%';
SELECT * FROM DBlocks
WHERE spid = @@spid
AND entity_name = 'Product';
COMMIT TRAN
```

RESULTS FROM *DBlocks*

```
spid dbname              entity_name  index_id   resource   description     mode  status
---- ------------------- -----------  ---------- ---------- --------------  ----- -------
54   AdventureWorks2008  Product      NULL       OBJECT                     IX    GRANT
54   AdventureWorks2008  Product      1          PAGE       1:16897         IX    GRANT
54   AdventureWorks2008  Product      1          KEY        (6b00b8eeda30)  X     GRANT
54   AdventureWorks2008  Product      1          KEY        (6a00dd8966 88) X     GRANT
```

The two rows in the leaf level of the clustered index are locked with X locks. The page and the table are then locked with IX locks. I mentioned earlier that SQL Server actually acquires update locks while it looks for the rows to update. However, these are converted to X locks when the actual update is performed, and by the time we look at the *DBLocks* view, the update locks are gone. Unless you actually force update locks with a query hint, you might never see them in the lock report from *DBLocks* or by direct inspection of *sys.dm_tran_locks*.

Example 5: Update with Serializable Isolation Level Using an Index

SQL BATCH

```
USE AdventureWorks2008;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRAN
UPDATE Production.Product
SET ListPrice = ListPrice * 0.6
WHERE Name LIKE 'Racing Socks%';
SELECT * FROM DBlocks
WHERE spid = @@spid
AND entity_name = 'Product';
COMMIT TRAN
```

RESULTS FROM *DBlocks*

```
spid dbname              entity_name  index_id    resource   description     mode     status
---- ------------------- ------------ ----------- ---------- --------------- -------- ------
54   AdventureWorks2008  Product      NULL        OBJECT                     IX       GRANT
54   AdventureWorks2008  Product      1           PAGE       1:16897         IX       GRANT
54   AdventureWorks2008  Product      1           KEY        (6a00dd896688)  X        GRANT
54   AdventureWorks2008  Product      1           KEY        (6b00b8eeda30)  X        GRANT
54   AdventureWorks2008  Product      3           KEY        (9502d56a217e)  RangeS-U GRANT
54   AdventureWorks2008  Product      3           PAGE       1:1767          IU       GRANT
54   AdventureWorks2008  Product      3           KEY        (23027a50f6db)  RangeS-U GRANT
54   AdventureWorks2008  Product      3           KEY        (9602945b3a67)  RangeS-U GRANT
```

Again, notice that the key-range locks are on the nonclustered index used to find the relevant rows. The range interval itself needs only a shared lock to prevent insertions, but the searched keys have U locks so no other process can attempt to update them. The keys in the table itself (*index_id* = 1) obtain the exclusive lock when the actual modification is made.

Now let's look at an *UPDATE* operation with the same isolation level when no index can be used for the search.

Example 6: Update with Serializable Isolation Not Using an Index

SQL BATCH

```
USE AdventureWorks2008;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRAN
UPDATE Production.Product
SET ListPrice = ListPrice * 0.6
WHERE Color = 'White';
SELECT * FROM DBlocks
WHERE spid = @@spid
AND entity_name = 'Product';
COMMIT TRAN
```

RESULTS FROM *DBlocks* (Abbreviated)

```
spid dbname              entity_name  index_id    resource   description    mode     status
---- ------------------- ------------ ----------- ---------- -------------- -------- -------
54   AdventureWorks2008  Product      NULL        OBJECT                    IX       GRANT
54   AdventureWorks2008  Product      1           KEY        (7900ac71caca) RangeS-U GRANT
54   AdventureWorks2008  Product      1           KEY        (6100dc0e675f) RangeS-U GRANT
54   AdventureWorks2008  Product      1           KEY        (5700a1a9278a) RangeS-U GRANT
54   AdventureWorks2008  Product      1           PAGE       1:16898        IU       GRANT
54   AdventureWorks2008  Product      1           PAGE       1:16899        IU       GRANT
54   AdventureWorks2008  Product      1           PAGE       1:16896        IU       GRANT
54   AdventureWorks2008  Product      1           PAGE       1:16897        IX       GRANT
54   AdventureWorks2008  Product      1           PAGE       1:16900        IU       GRANT
54   AdventureWorks2008  Product      1           PAGE       1:16901        IU       GRANT
54   AdventureWorks2008  Product      1           KEY        (5600c4ce9b32) RangeS-U GRANT
54   AdventureWorks2008  Product      1           KEY        (7300c89177a5) RangeS-U GRANT
54   AdventureWorks2008  Product      1           KEY        (7f00702ea1ef) RangeS-U GRANT
54   AdventureWorks2008  Product      1           KEY        (6b00b8eeda30) RangeX-X GRANT
54   AdventureWorks2008  Product      1           KEY        (c500b9eaac9c) RangeX-X GRANT
54   AdventureWorks2008  Product      1           KEY        (c6005745198e) RangeX-X GRANT
54   AdventureWorks2008  Product      1           KEY        (6a00dd896688) RangeX-X GRANT
```

The locks here are similar to those in the previous example except that all the locks are on the table itself (*index_id* = 1). A clustered index scan (on the entire table) had to be done, so all keys initially received the RangeS-U lock, and when four rows were eventually modified, the locks on those keys were converted to RangeX-X locks. You can see all the RangeX-X locks, but not all the RangeS-U locks are shown for space reasons (the table has 504 rows).

Example 7: Creating a Table

SQL BATCH

```
USE AdventureWorks2008;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN TRAN
SELECT *
INTO newProducts
FROM Production.Product
WHERE ListPrice between 1 and 10;
SELECT * FROM DBlocks
WHERE spid = @@spid;
COMMIT TRAN
```

RESULTS FROM *DBlocks* (Abbreviated)

spid	dbname	entity_name	index_id	resource	description	mode	status
54	AdventureWorks2008	n/a	NULL	DATABASE		NULL	GRANT
54	AdventureWorks2008	n/a	NULL	DATABASE		NULL	GRANT
54	AdventureWorks2008	n/a	NULL	DATABASE		S	GRANT
54	AdventureWorks2008	n/a	NULL	METADATA	user_type_id = 258	Sch-S	GRANT
54	AdventureWorks2008	n/a	NULL	METADATA	data_space_id = 1	Sch-S	GRANT
54	AdventureWorks2008	n/a	NULL	DATABASE		S	GRANT
54	AdventureWorks2008	n/a	NULL	METADATA	$seq_type = 0, objec	Sch-M	GRANT
54	AdventureWorks2008	n/a	NULL	METADATA	user_type_id = 260	Sch-S	GRANT
54	AdventureWorks2008	sysrowsetcol	NULL	OBJECT		IX	GRANT
54	AdventureWorks2008	sysrowsets	NULL	OBJECT		IX	GRANT
54	AdventureWorks2008	sysallocunit	NULL	OBJECT		IX	GRANT
54	AdventureWorks2008	syshobtcolum	NULL	OBJECT		IX	GRANT
54	AdventureWorks2008	syshobts	NULL	OBJECT		IX	GRANT
54	AdventureWorks2008	sysserefs	NULL	OBJECT		IX	GRANT
54	AdventureWorks2008	sysschobjs	NULL	OBJECT		IX	GRANT
54	AdventureWorks2008	syscolpars	NULL	OBJECT		IX	GRANT
54	AdventureWorks2008	sysidxstats	NULL	OBJECT		IX	GRANT
54	AdventureWorks2008	sysrowsetcol	1	KEY	(15004f6b3486)	X	GRANT
54	AdventureWorks2008	sysrowsetcol	1	KEY	(0a00862c4e8e)	X	GRANT
54	AdventureWorks2008	sysrowsets	1	KEY	(000000aaec7b)	X	GRANT
54	AdventureWorks2008	sysallocunit	1	KEY	(00001f2dcf47)	X	GRANT
54	AdventureWorks2008	syshobtcolum	1	KEY	(1900f7d4e2cc)	X	GRANT
54	AdventureWorks2008	syshobts	1	KEY	(000000aaec7b)	X	GRANT
54	AdventureWorks2008	NULL	NULL	RID	1:6707:1	X	GRANT
54	AdventureWorks2008	DBlocks	NULL	OBJECT		IS	GRANT
54	AdventureWorks2008	newProducts	NULL	OBJECT		Sch-M	GRANT
54	AdventureWorks2008	sysserefs	1	KEY	(010025fabf73)	X	GRANT
54	AdventureWorks2008	sysschobjs	1	KEY	(3b0042322c99)	X	GRANT
54	AdventureWorks2008	syscolpars	1	KEY	(4200c1eb801c)	X	GRANT
54	AdventureWorks2008	syscolpars	1	KEY	(4e00092bfbc3)	X	GRANT
54	AdventureWorks2008	sysidxstats	1	KEY	(3b0006e110a6)	X	GRANT
54	AdventureWorks2008	sysschobjs	2	KEY	(9202706f3e6c)	X	GRANT
54	AdventureWorks2008	syscolpars	2	KEY	(6c0151be80af)	X	GRANT
54	AdventureWorks2008	syscolpars	2	KEY	(2c03557a0b9d)	X	GRANT
54	AdventureWorks2008	sysidxstats	2	KEY	(3c00f3332a43)	X	GRANT
54	AdventureWorks2008	sysschobjs	3	KEY	(9202d42ddd4d)	X	GRANT
54	AdventureWorks2008	sysschobjs	4	KEY	(3c0040d00163)	X	GRANT
54	AdventureWorks2008	newProducts	0	PAGE	1:6707	X	GRANT
54	AdventureWorks2008	newProducts	0	HOBT		Sch-M	GRANT

Very few of these locks are actually acquired on elements of the *newProducts* table. In the *entity_name* column, you can see that most of the objects are undocumented, and normally invisible, system table names. As the new table is created, SQL Server acquires locks on nine different system tables to record information about this new table. In addition, notice the schema modification (Sch-M) lock and other metadata locks on the new table.

The final example looks at the locks held when there is no clustered index on the table and the data rows are being updated.

Example 8: Row Locks

SQL BATCH

```
USE AdventureWorks2008;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
BEGIN TRAN
UPDATE newProducts
SET ListPrice = 5.99
WHERE name = 'Road Bottle Cage';
SELECT * FROM DBlocks
WHERE spid = @@spid
AND entity_name = 'newProducts';
COMMIT TRAN
```

RESULTS FROM *DBlocks*

```
spid dbname              entity_name  index_id    resource   description  mode     status
---- ------------------- ------------ ----------- ---------- ------------ -------- --------
54   AdventureWorks2008  newProducts  NULL        OBJECT                  IX       GRANT
54   AdventureWorks2008  newProducts  0           PAGE       1:6708       IX       GRANT
54   AdventureWorks2008  newProducts  0           RID        1:6708:5     X        GRANT
```

There are no indexes on the *newProducts* table, so the lock on the actual row meeting our criteria is an exclusive (X) lock on the row (RID). For RID locks, the description actually reports the specific row in the form *File number:Page number:Slot number*. As expected, IX locks are taken on the page and the table.

Lock Compatibility

Two locks are compatible if one lock can be granted while another lock on the same resource is held by a different process. If a lock requested for a resource is not compatible with a lock currently being held, the requesting connection must wait for the lock. For example, if a shared page lock exists on a page, another process requesting a shared page lock for the same page is granted the lock because the two lock types are compatible. But a process that requests an exclusive lock for the same page is not granted the lock because an exclusive lock is not compatible with the shared lock already held. Figure 10-2 summarizes the compatibility of locks in SQL Server 2008. Along the top are all the lock modes that a process might already hold. Along the left edge are the lock modes that another process might request.

At the point where the held lock and requested lock meet, there can be three possible values. *N* indicates that there is no conflict, *C* indicates that there will be a conflict and the requesting process will have to wait, and *I* indicates an invalid combination that could never occur. All the *I* values in the chart involve range locks, which can be applied only to KEY resources, so any type of lock that can never be applied to KEY resources indicates an invalid comparison.

	NL	SCH-S	SCH-M	S	U	X	IS	IU	IX	SIU	SIX	UIX	BU	RS-S	RS-U	RI-N	RI-S	RI-U	RI-X	RX-S	RX-U	RX-X
NL	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
SCH-S	N	N	C	N	N	N	N	N	N	N	N	N	N	I	I	I	I	I	I	I	I	I
SCH-M	N	C	C	C	C	C	C	C	C	C	C	C	C	I	I	I	I	I	I	I	I	I
S	N	N	C	N	N	C	N	N	C	N	C	C	C	N	N	N	N	N	C	N	N	C
U	N	N	C	N	C	C	N	C	C	C	C	C	C	N	C	N	N	C	C	N	C	C
X	N	N	C	C	C	C	C	C	C	C	C	C	C	C	C	N	C	C	C	C	C	C
IS	N	N	C	N	N	C	N	N	N	N	N	N	C	I	I	I	I	I	I	I	I	I
IU	N	N	C	N	C	C	N	N	N	N	N	C	C	I	I	I	I	I	I	I	I	I
IX	N	N	C	C	C	C	N	N	N	C	C	C	C	I	I	I	I	I	I	I	I	I
SIU	N	N	C	N	C	C	N	N	C	N	C	C	C	I	I	I	I	I	I	I	I	I
SIX	N	N	C	C	C	C	N	N	C	C	C	C	C	I	I	I	I	I	I	I	I	I
UIX	N	N	C	C	C	C	N	C	C	C	C	C	C	I	I	I	I	I	I	I	I	I
BU	N	N	C	C	C	C	C	C	C	C	C	C	N	I	I	I	I	I	I	I	I	I
RS-S	N	I	I	N	N	C	I	I	I	I	I	I	I	N	N	C	C	C	C	C	C	C
RI-U	N	I	I	N	C	C	I	I	I	I	I	I	I	N	C	C	C	C	C	C	C	C
RI-N	N	I	I	N	N	N	I	I	I	I	I	I	I	C	C	N	N	N	N	C	C	C
RI-S	N	I	I	N	N	C	I	I	I	I	I	I	I	C	C	N	N	N	C	C	C	C
RI-U	N	I	I	N	C	C	I	I	I	I	I	I	I	C	C	N	N	C	C	C	C	C
RI-X	N	I	I	C	C	C	I	I	I	I	I	I	I	C	C	N	C	C	C	C	C	C
RX-S	N	I	I	N	N	C	I	I	I	I	I	I	I	C	C	C	C	C	C	C	C	C
RX-U	N	I	I	N	C	C	I	I	I	I	I	I	I	C	C	C	C	C	C	C	C	C
RX-X	N	I	I	C	C	C	I	I	I	I	I	I	I	C	C	C	C	C	C	C	C	C

FIGURE 10-2 SQL Server lock compatibility matrix

Lock compatibility comes into play between locks on different resources, such as table locks and page locks. A table and a page obviously represent an implicit hierarchy because a table is made up of multiple pages. If an exclusive page lock is held on one page of a table, another process cannot get even a shared table lock for that table. This hierarchy is protected using intent locks. A process acquiring an exclusive page lock, update page lock, or intent exclusive page lock first acquires an intent exclusive lock on the table. This intent exclusive table lock prevents another process from acquiring the shared table lock on that table. (Remember that intent exclusive locks and shared locks on the same resource are not compatible.)

Similarly, a process acquiring a shared row lock must first acquire an intent shared lock for the table, which prevents another process from acquiring an exclusive table lock. Or if the exclusive table lock already exists, the intent shared lock is not granted and the shared page lock has to wait until the exclusive table lock is released. Without intent locks, process A can lock a page in a table with an exclusive page lock and process B can place an exclusive table lock on the same table and hence think that it has a right to modify the entire table, including the page that process A has exclusively locked.

Note Obviously, lock compatibility is an issue only when the locks affect the same object. For example, two or more processes each can hold exclusive page locks simultaneously so long as the locks are on different pages or different tables.

Even if two locks are compatible, the requester of the second lock might still have to wait if an incompatible lock is waiting. For example, suppose that process A holds a shared page lock. Process B requests an exclusive page lock and must wait because the shared page lock and the exclusive page lock are not compatible. Process C requests a shared page lock

that is compatible with the shared page already granted to process A. However, the shared page lock cannot be granted immediately. Process C must wait for its shared page lock because process B is ahead of it in the lock queue with a request (exclusive page) that is not compatible.

By examining the compatibility of locks not only with processes granted locks, but also processes waiting, SQL Server prevents lock starvation, which can result when requests for shared locks keep overlapping so that the request for the exclusive lock can never be granted.

Internal Locking Architecture

Locks are not on-disk structures. You won't find a lock field directly on a data page or a table header, and the metadata that keeps track of locks is never written to disk. Locks are internal memory structures—they consume part of the memory used for SQL Server. A lock is identified by *lock resource,* which is a description of the resource that is locked (a row, index key, page, or table). To keep track of the database, the type of lock, and the information describing the locked resource, each lock requires 64 bytes of memory on a 32-bit system and 128 bytes of memory on a 64-bit system. This 64-byte or 128-byte structure is called a *lock block*.

Each process holding a lock also must have a *lock owner,* which represents the relationship between a lock and the entity that is requesting or holding the lock. The lock owner requires 32 bytes of memory on a 32-bit system and 64 bytes of memory on a 64-bit system. This 32-byte or 64-byte structure is called a *lock owner block*. A single transaction can have multiple lock owner blocks; a scrollable cursor sometimes uses several. Also, one lock can have many lock owner blocks, as is the case with a shared lock. As mentioned, the lock owner represents a relationship between a lock and an entity, and the relationship can be granted, waiting, or in a state called *waiting-to-convert*.

The lock manager maintains a lock hash table. Lock resources, contained within a lock block, are hashed to determine a target hash slot in the hash table. All lock blocks that hash to the same slot are chained together from one entry in the hash table. Each lock block contains a 15-byte field that describes the locked resource. The lock block also contains pointers to lists of lock owner blocks. There is a separate list for lock owners in each of the three states. Figure 10-3 shows the general lock architecture.

The number of slots in the hash table is based on the system's physical memory, as shown in Table 10-7. There is an upper limit of 2^{31} slots. All instances of SQL Server on the same machine have a hash table with the same number of slots. Each entry in the lock hash table is 16 bytes in size and consists of a pointer to a list of lock blocks and a spinlock to guarantee serialized access to the same slot.

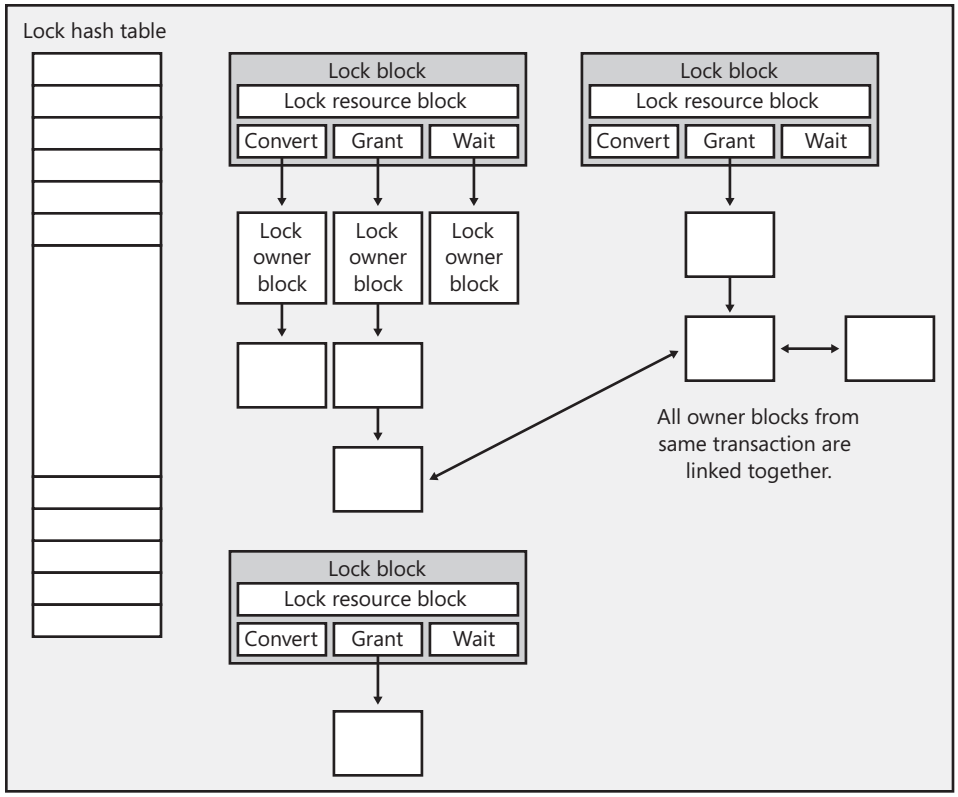


FIGURE 10-3 SQL Server locking architecture

TABLE 10-7 Number of Slots in the Internal Lock Hash Table

Physical Memory (MB)	Number of Slots	Memory Used
< 32	2^{14} = 16384	128 KB
>= 32 and < 64	2^{15} = 32768	256 KB
>= 64 and < 128	2^{16} = 65536	512 KB
>= 128 and < 512	2^{18} = 262144	2048 KB
>= 512 and < 1024	2^{19} = 524288	4096 KB
>= 1024 and < 4096	2^{21} = 2097152	16384 KB
>= 4096 and < 8192	2^{22} = 4194304	32768 KB
>= 8192 and < 16384	2^{23} = 8388608	65536 KB
>= 16384	2^{25} = 33554432	262144 KB

The lock manager allocates in advance a number of lock blocks and lock owner blocks at server startup. On NUMA configurations, these lock and lock owner blocks are divided among all NUMA nodes. So when a lock request is made, local lock blocks are used. If the number of locks has been set by *sp_configure,* it allocates that configured number of lock

blocks and the same number of lock owner blocks. If the number is not fixed (0 means auto-tune), it allocates 2,500 lock blocks for your SQL Server instance. It allocates twice as many (2 * # lock blocks) of the lock owner blocks. At their maximum, the static allocations can't consume more than 25 percent of the committed buffer pool size.

When a request for a lock is made and no free lock blocks remain, the lock manager dynamically allocates new lock blocks instead of denying the lock request. The lock manager cooperates with the global memory manager to negotiate for server allocated memory. When necessary, the lock manager can free the dynamically allocated lock blocks. The lock manager is limited to 60 percent of the buffer manager's committed target size allocation to lock blocks and lock owner blocks.

Lock Partitioning

For large systems, locks on frequently referenced objects can become a performance bottleneck. The process of acquiring and releasing locks can cause contention on the internal locking resources. Lock partitioning enhances locking performance by splitting a single lock resource into multiple lock resources. For systems with 16 or more CPUs, SQL Server automatically splits certain locks into multiple lock resources, one per CPU. This is called *lock partitioning,* and there is no way for a user to control this process. (Do not confuse lock partitioning with partition locks, which are discussed in the section entitled "Lock Escalation," later in this chapter.) An informational message is sent to the error log whenever lock partitioning is active. The error message is "Lock partitioning is enabled. This is an informational message only. No user action is required." Lock partitioning applies only to full object locks (for example, tables and views) in the following lock modes: S, U, X, and SCH-M. All other modes (NL, SCH_S, IS, IU, and IX) are acquired on a single CPU. SQL Server assigns a default lock partition to every transaction when the transaction starts. During the life of that transaction, all lock requests that are spread over all the partitions use the partition assigned to that transaction. By this method, access to lock resources of the same object by different transactions is distributed across different partitions.

The *resource_lock_partition* column in *sys.dm_tran_locks* indicates which lock partition a particular lock is on, so you can see multiple locks for the exact same resource with different *resource_lock_partition* values. For systems with fewer than 16 CPUs, for which lock partitioning is never used, the *resource_lock_partition* value is always 0.

For example, consider a transaction acquiring an IS lock in REPEATABLE READ isolation, so that the IS lock is held for the duration of the transaction. The IS lock is acquired on the transaction's default partition—for example, partition 4. If another transaction tries to acquire an X lock on the same table, the X lock must be acquired on ALL partitions. SQL Server successfully acquires the X lock on partitions 0 to 3, but it blocks when attempting to acquire an X lock on partition 4. On partition IDs 5 to 15, which have not yet acquired the X lock for this table, other transactions can continue to acquire any locks that do not cause blocking.

With lock partitioning, SQL Server distributes the load of checking for locks across multiple spinlocks, and most accesses to any given spinlock are from the same CPU (and practically always from the same node), which means the spinlock should not spin often.

Lock Blocks

The lock block is the key structure in SQL Server's locking architecture, shown earlier in Figure 10-3. A lock block contains the following information:

- Lock resource information containing the lock resource name and details about the lock.
- Pointers to connect the lock blocks to the lock hash table.
- Pointers to lists of lock owner blocks for locks on this resource that have been granted. Four *grant lists* are maintained to minimize the amount of time it takes to find a granted lock.
- A pointer to a list of lock owner blocks for locks on this resource that are waiting to be converted to another lock mode. This is called the *convert list*.
- A pointer to a list of lock owner blocks for locks that have been requested on this resource but have not yet been granted. This is called the *wait list*.

The lock resource uniquely identifies the data being locked. Its structure is shown in Figure 10-4. Each "row" in the figure represents 4 bytes, or 32 bits.

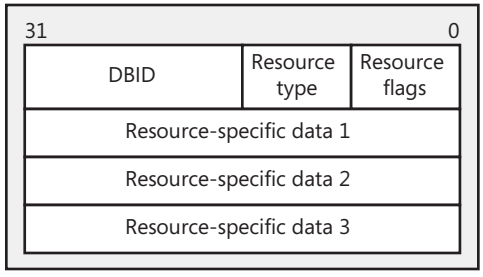


FIGURE 10-4 The structure of a lock resource

The meanings of the fields shown in Figure 10-4 are described in Table 10-8. The value in the *resource type* byte is one of the locking resources described earlier in Table 10-5. The number in parentheses after the resource type is the code number for the resource type (which we see in the *syslockinfo* table a little later in the chapter). The meaning of the values in the three data fields varies depending on the type of resource being described. SR indicates a subresource (which I describe shortly).

TABLE 10-8 Fields in the Lock Resource Block

	Resource Content		
Resource Type	**Data 1**	**Data 2**	**Data 3**
Database (2)	SR	0	0
File (3)	File ID	0	0
Index (4)	Object ID	SR	Index ID
Table (5)	Object ID	SR	0
Page (6)	Page number		0
Key (7)	Partition ID	Hashed key	
Extent (8)	Extent ID		0
RID (9)	RID		0

The following are some of the possible SR (SubResource) values. If the lock is on a Database resource, SR indicates one of the following:

- Full database lock
- Bulk operation lock

If the lock is on a Table resource, SR indicates one of the following:

- Full table lock (default)
- Update statistics lock
- Compile lock

If the lock is on an Index resource, SR indicates one of the following:

- Full index lock (default)
- Index ID lock
- Index name lock

Lock Owner Blocks

Each lock owned or waited for by a session is represented in a lock owner block. Lists of lock owner blocks form the grant, convert, and wait lists that hang off the lock blocks. Each lock owner block for a granted lock is linked with all other lock owner blocks for the same transaction or session so they can be freed as appropriate when the transaction or session ends.

syslockinfo Table

Although the recommended way of retrieving information about locks is through the *sys.dm_tran_locks* view, there is another metadata object called *syslockinfo* that provides internal information about locks. Prior to the introduction of the DMVs in SQL Server 2005, *syslockinfo* was the only internal metadata available for examining locking information.

In fact, the stored procedure *sp_lock* is still defined to retrieve information from *syslockinfo* instead of from *sys.dm_tran_locks*. I will not go into full detail about *syslockinfo* because almost all the information from that table is available, in a much more readable form, in the *sys.dm_tran_locks* view. However, *syslockinfo* is available in the *master* database for you to take a look at. One column, however, is of particular interest—the *rsc_bin* column, which contains a 16-byte description of a locked resource.

You can analyze the *syslockinfo.rsc_bin* field as the resource block. Let's look at an example. I select a single row from the *Person* table in *AdventureWorks2008* using the REPEATABLE READ isolation level, so my shared locks continue to be held for the duration of the transaction. I then look at the *rsc_bin* column in *syslockinfo* for key locks, page locks, and table locks:

```
USE AdventureWorks2008
GO
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
GO
BEGIN TRAN
SELECT * FROM Person.Person
WHERE BusinessEntityID = 249;
GO
SELECT rsc_bin, rsc_type
FROM master..syslockinfo
WHERE rsc_type IN (5,6,7);
GO
```

Here are the three rows in the result set:

```
rsc_bin                            rsc_type
---------------------------------- --------
0x805EFA590000000000000000007000500 5
0x190500000100000000000000007000600 6
0x710000000001F900CE79D52507000700 7
```

The last 2 bytes in *rsc_bin* are the resource mode, so after byte-swapping, you can see the same value as in the *rsc_type* column—for example, you byte-swap 0500 to 0005 to resource mode 5 (a table lock). The next 2 bytes at the end indicate the database ID, and for all three rows, the value after byte-swapping is 0007, which is the database ID of my *AdventureWorks2008* database.

The rest of the bytes vary depending on the type of resource. For a table, the first 4 bytes represent the object ID. The preceding row for the object lock (*rsc_type* = 5) after byte swapping has a value of 59FA5E80, which is 1509580416 in decimal. I can translate this to an object name as follows:

```
SELECT object_name(1509580416)
```

This shows me the *Person* table.

For a PAGE (*rsc_type* = 6), the first 6 bytes are the page number followed by the file number. After byte-swapping, the file number is 0001, or 1 decimal, and the page number is 00000519, or 9889 in decimal. So the lock is on file 1, page 1305.

Finally, for a KEY (*rsc_type* = 7), the first 6 bytes represent the partition ID but the translation is a bit trickier. We need to add another 2 bytes of zeros to the value after byte-swapping, so we end up with 0100000000710000, which translates to 72057594045333504 in decimal. To see which object this partition belongs to, I can query the *sys.partitions* view:

```
SELECT object_name(object_id)
FROM sys.partitions
WHERE partition_ID = 72057594045333504;
```

Again, the result is that this partition is part of the *Person* table. The next 6 bytes of *rsc_bin* for the KEY resource are F900CE79D525. This is a character field, so no byte-swapping is needed. However, the value is not further decipherable. Key locks have a hash value generated for them, based on all the key columns of the index. Indexes can be quite long, so for almost any possible data type, SQL Server needs a consistent way to keep track of which keys are locked. The hashing function therefore generates a 6-byte hash string to represent the key. Although you can't reverse-engineer this value and determine exactly which index row is locked, you can use it to look for matching entries, just like SQL Server does. If two *rsc_bin* values have the same 6-byte hash string, they are referring to the same lock resource.

In addition to detecting references to the same lock resource, you can determine which specific keys are locked by using the undocumented value *%%lockres%%*, which can return the hash string for any key. Selecting this value, along with data from the table, returns the lock resource for every row in the result set, based on the index used to retrieve the data. Consider the following example, which creates a clustered and nonclustered index on a tiny table and then selects the *%%lockres%%* value for each row, first using the clustered index and then using the nonclustered index:

```
CREATE TABLE lockres (c1 int, c2 int);
GO
INSERT INTO lockres VALUES (1,10);
INSERT INTO lockres VALUES (2,20);
INSERT INTO lockres VALUES (3,30);
GO
CREATE UNIQUE CLUSTERED INDEX lockres_ci ON lockres(c1);
CREATE UNIQUE NONCLUSTERED INDEX lockres_nci ON lockres(c2);
GO
SELECT %%lockres%% AS lock_resource, * FROM lockres WITH (INDEX = lockres_ci);
SELECT %%lockres%% AS lock_resource, * FROM lockres WITH (INDEX = lockres_nci);
GO
```

I get the following results. The first set of rows shows the lock resource for the clustered index keys, and the second set shows the lock resources for the nonclustered index:

```
lock_resource                  c1          c2
------------------------------ ----------- -----------
(010086470766)                 1           10
(020068e8b274)                 2           20
(03000d8f0ecc)                 3           30
```

```
lock_resource                    c1          c2
-------------------------------- ----------- -----------
(0a0087c006b1)                   1           10
(14002be0c001)                   2           20
(1e004f007d6e)                   3           30
```

I can use this lock resource to find which row in a table matches a locked resource. For example, if *sys.dm_tran_locks* indicates that a row with the lock resource (010086470766) is holding a lock in the *lockres* table, I could find which row that resource corresponds to with the following query:

```
SELECT * FROM lockres
WHERE %%lockres%% = '(010086470766)'
```

Note that if the table is a heap and I look for the lock resource when scanning the table, the lock resource is the actual row ID (RID). The value returned looks just like the special value *%%physloc%%,* which I told you about in Chapter 5, "Tables":

```
CREATE TABLE lockres_on_heap (c1 int, c2 int);
GO
INSERT INTO lockres_on_heap VALUES (1,10);
INSERT INTO lockres_on_heap VALUES (2,20);
INSERT INTO lockres_on_heap VALUES (3,30);
GO
SELECT %%lockres%% AS lock_resource, * FROM lockres_on_heap;
```

Here are my results:

```
lock_resource                    c1          c2
-------------------------------- ----------- ----
1:169:0                          1           10
1:169:1                          2           20
1:169:2                          3           30
```

Caution You need to be careful when trying to find the row in a table with a hash string that matches a particular lock resource. These queries have to perform a complete scan of the table to find the row you are interested in, and with a large table, that process can be very expensive.

Row-Level Locking vs. Page-Level Locking

Although SQL Server 2008 fully supports row-level locking, in some situations, the lock manager decides not to lock individual rows and instead locks pages or the whole table. In other cases, many smaller locks are escalated to a table lock, as I discuss in the upcoming section entitled "Lock Escalation."

Prior to SQL Server 7.0, the smallest unit of data that SQL Server could lock was a page. Even though many people argued that this was unacceptable and it was impossible to maintain good concurrency while locking entire pages, many large and powerful applications were written

and deployed using only page-level locking. If they were well designed and tuned, concurrency was not an issue, and some of these applications supported hundreds of active user connections with acceptable response times and throughput. However, with the change in page size from 2 KB to 8 KB for SQL Server 7.0, the issue has become more critical. Locking an entire page means locking four times as much data as in previous versions. Beginning with SQL Server 7.0, the software implements full row-level locking, so any potential problems due to lower concurrency with the larger page size should not be an issue. However, locking isn't free. Resources are required to manage locks. Recall that a lock is an in-memory structure of 64 or 128 bytes (for 32-bit or 64-bit machines, respectively) with another 32 or 64 bytes for each process holding or requesting the lock. If you need a lock for every row and you scan a million rows, you need more than 64 MB of RAM just to hold locks for that one process.

Beyond memory consumption issues, locking is a fairly processing-intensive operation. Managing locks requires substantial bookkeeping. Recall that, internally, SQL Server uses a lightweight mutex called a *spinlock* to guard resources, and it uses latches—also lighter than full-blown locks—to protect non-leaf level index pages. These performance optimizations avoid the overhead of full locking. If a page of data contains 50 rows of data, all of which are used, it is obviously more efficient to issue and manage one lock on the page than to manage 50. That's the obvious benefit of page locking—a reduction in the number of lock structures that must exist and be managed.

Let's say two processes each need to update a few rows of data, and even though the rows are not the same ones, some of them happen to exist on the same page. With page-level locking, one process would have to wait until the page locks of the other process were released. If you use row-level locking instead, the other process does not have to wait. The finer granularity of the locks means that no conflict occurs in the first place because each process is concerned with different rows. That's the obvious benefit of row-level locking. Which of these obvious benefits wins? Well, the decision isn't clear-cut, and it depends on the application and the data. Each type of locking can be shown to be superior for different types of applications and usage.

The *ALTER INDEX* statement lets you manually control the unit of locking within an index with options to disallow page locks or row locks within an index. Because these options are available only for indexes, there is no way to control the locking within the data pages of a heap. (But remember that if a table has a clustered index, the data pages are part of the index and are affected by a value set with *ALTER INDEX*.) The index options are set for each table or index individually. Two options, ALLOW_ROW_LOCKS and ALLOW_PAGE_LOCKS, are both set to ON initially for every table and index. If both of these options are set to OFF for a table, only full table locks are allowed.

As mentioned earlier, during the optimization process, SQL Server determines whether to lock rows, pages, or the entire table initially. The locking of rows (or keys) is heavily favored. The type of locking chosen is based on the number of rows and pages to be scanned, the number of rows on a page, the isolation level in effect, the update activity going on, the number of users on the system needing memory for their own purposes, and so on.

Lock Escalation

SQL Server automatically escalates row, key, or page locks to coarser table or partition locks as appropriate. This escalation protects system resources—it prevents the system from using too much memory for keeping track of locks—and increases efficiency. For example, after a query acquires many row locks, the lock level can be escalated because it probably makes more sense to acquire and hold a single lock than to hold many row locks. When lock escalation occurs, many locks on smaller units (rows or pages) are released and replaced by one lock on a larger unit. This escalation reduces locking overhead and keeps the system from running out of locks. Because a finite amount of memory is available for the lock structures, escalation is sometimes necessary to make sure the memory for locks stays within reasonable limits.

The default in SQL Server is to escalate to table locks. However, SQL Server 2008 introduces the ability to escalate to a single partition using the *ALTER TABLE* statement. The LOCK_ESCALATION option of *ALTER TABLE* can specify that escalation is always to a table level, or that it can be to either a table or partition level. The LOCK_ESCALATION option can also be used to prevent escalation entirely. Here's an example of altering the *TransactionHistory* table (which you may have created if you ran the partitioning example in Chapter 7, "Special Storage"), so that locks can be escalated to either the table or partition level:

```
ALTER TABLE TransactionHistory
SET (LOCK_ESCALATION = AUTO);
```

Lock escalation occurs in the following situations:

- The number of locks held by a single statement on one object, or on one partition of one object, exceeds a threshold. Currently that threshold is 5,000 locks, but it might change in future service packs. The lock escalation does not occur if the locks are spread over multiple objects in the same statement—for example, 3,000 locks in one index and 3,000 in another.
- Memory taken by lock resources exceeds 40 percent of the non-AWE (32-bit) or regular (64-bit) enabled memory and the locks configuration option is set to 0. (In this case, the lock memory is allocated dynamically as needed, so the 40 percent value is not a constant.) If the locks option is set to a nonzero value, memory reserved for locks is statically allocated when SQL Server starts. Escalation occurs when SQL Server is using more than 40 percent of the reserved lock memory for lock resources.

When the lock escalation is triggered, the attempt might fail if there are conflicting locks. So, for example, if an X lock on a RID needs to be escalated and there are concurrent X locks on the same table or partition held by a different process, the lock escalation attempt fails. However, SQL Server continues to attempt to escalate the lock every time the transaction acquires another 1,250 locks on the same object. If the lock escalation succeeds, SQL Server releases all the row and page locks on the index or the heap.

Note SQL Server never escalates to page locks. The result of a lock escalation is always a table or partition. In addition, multiple partition locks are never escalated to a table lock.

Controlling Lock Escalation

Lock escalation can potentially lead to blocking of future concurrent access to the index or the heap by other transactions needing row or page locks on the object. SQL Server cannot de-escalate the lock when new requests are made. So lock escalation is not always a good idea for all applications.

SQL Server 2008 also supports disabling lock escalation for a single table using the *ALTER TABLE* statement. Here is an example of disabling lock escalation on the *TransactionHistory* table:

```
ALTER TABLE TransactionHistory
SET (LOCK_ESCALATION = DISABLE);
```

SQL Server 2008 also supports disabling lock escalation using trace flags. Note that these trace flags affect lock escalation on all tables in all databases in a SQL Server instance.

- Trace flag 1211 completely disables lock escalation. It instructs SQL Server to ignore the memory acquired by the lock manager up to the maximum statically allocated lock memory (specified using the locks configuration option) or 60 percent of the non-AWE (32-bit) or regular (64-bit) dynamically allocated memory. At that time, an out-of-lock memory error is generated. You should exercise extreme caution when using this trace flag as a poorly designed application can exhaust the memory and seriously degrade the performance of your SQL Server instance.
- Trace flag 1224 also disables lock escalation based on the number of locks acquired, but it allows escalation based on memory consumption. It enables lock escalation when the lock manager acquires 40 percent of the statically allocated memory (as per the locks option) or 40 percent of the non-AWE (32-bit) or regular (64-bit) dynamically allocated memory. You should note that if SQL Server cannot allocate memory for locks due to memory use by other components, the lock escalation can be triggered earlier. As with trace flag 1211, SQL Server generates an out-of-memory error when memory allocated to the lock manager exceeds the total statically allocated memory or 60 percent of non-AWE (32-bit) or regular (64-bit) memory for dynamic allocation.

If both trace flags (1211 and 1224) are set at the same time, trace flag 1211 takes precedence. Remember that these trace flags affect the entire SQL Server instance. In many cases, it is desirable to control the escalation threshold at the object level, so you should consider using the *ALTER TABLE* command when possible.

Deadlocks

A deadlock occurs when two processes are waiting for a resource and neither process can advance because the other process prevents it from getting the resource. A true deadlock is

a Catch-22 in which, without intervention, neither process can ever make progress. When a deadlock occurs, SQL Server intervenes automatically. I refer mainly to deadlocks acquired due to conflicting locks, although deadlocks can also be detected on worker threads, memory, and parallel query resources.

Note A simple wait for a lock is not a deadlock. When the process that's holding the lock completes, the waiting process can acquire the lock. Lock waits are normal, expected, and necessary in multiuser systems.

In SQL Server, two main types of deadlocks can occur: a cycle deadlock and a conversion deadlock. Figure 10-5 shows an example of a cycle deadlock. Process A starts a transaction, acquires an exclusive table lock on the *Product* table, and requests an exclusive table lock on the *PurchaseOrderDetail* table. Simultaneously, process B starts a transaction, acquires an exclusive lock on the *PurchaseOrderDetail* table, and requests an exclusive lock on the *Product* table. The two processes become deadlocked—caught in a "deadly embrace." Each process holds a resource needed by the other process. Neither can progress, and, without intervention, both would be stuck in deadlock forever. You can actually generate the deadlock in SQL Server Management Studio, as follows:

1. Open a query window, and change your database context to the *AdventureWorks2008* database. Execute the following batch for process A:

```
BEGIN TRAN
UPDATE  Production.Product
    SET ListPrice = ListPrice * 0.9
WHERE ProductID  = 922;
```

2. Open a second window, and execute this batch for process B:

```
BEGIN TRAN
UPDATE  Purchasing.PurchaseOrderDetail
    SET OrderQty = OrderQty + 200
    WHERE ProductID  = 922
    AND PurchaseOrderID = 499;
```

3. Go back to the first window, and execute this *UPDATE* statement:

```
UPDATE  Purchasing.PurchaseOrderDetail
    SET OrderQty = OrderQty - 200
    WHERE ProductID  = 922
    AND PurchaseOrderID = 499;
```

   At this point, the query should block. It is not deadlocked yet, however. It is waiting for a lock on the *PurchaseOrderDetail* table, and there is no reason to suspect that it won't eventually get that lock.

4. Go back to the second window, and execute this *UPDATE* statement:

```
UPDATE  Production.Product
    SET ListPrice = ListPrice * 1.1
    WHERE ProductID  = 922;
```

At this point, a deadlock occurs. The first connection never gets its requested lock on the *PurchaseOrderDetail* table because the second connection does not give it up until it gets a lock on the *Product* table. Because the first connection already has the lock on the *Product* table, we have a deadlock. One of the processes receives the following error message. (Of course, the actual process ID reported will probably be different.)

```
Msg 1205, Level 13, State 51, Line 1
Transaction (Process ID 57) was deadlocked on lock resources with another process and has
been chosen as the deadlock victim. Rerun the transaction.
```

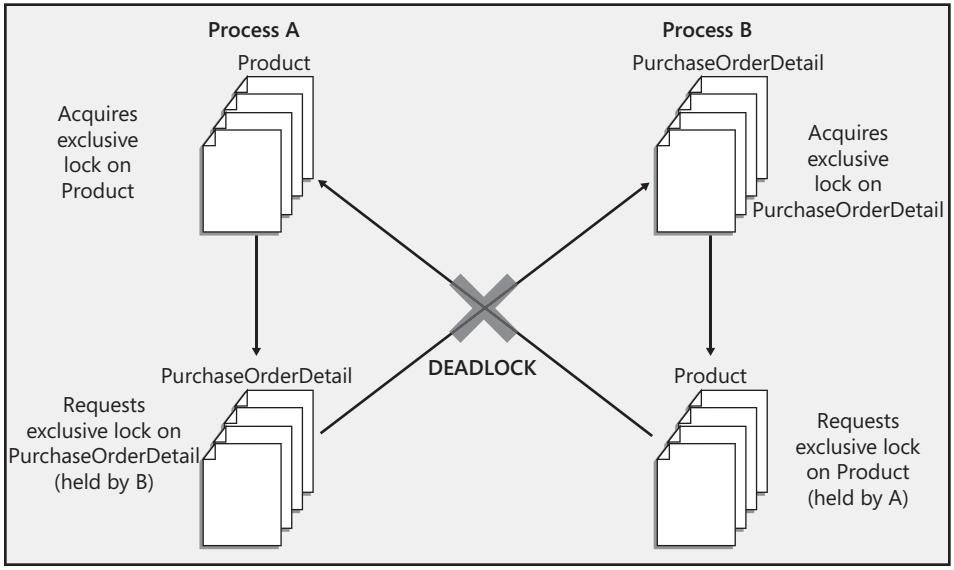


FIGURE 10-5 A cycle deadlock resulting from two processes, each holding a resource needed by the other

Figure 10-6 shows an example of a conversion deadlock. Process A and process B each hold a shared lock on the same page within a transaction. Each process wants to promote its shared lock to an exclusive lock but cannot do so because of the other process's lock. Again, intervention is required.

SQL Server automatically detects deadlocks and intervenes through the lock manager, which provides deadlock detection for regular locks. In SQL Server 2008, deadlocks can also involve resources other than locks. For example, if process A is holding a lock on *Table1* and is waiting for memory to become available and process B has some memory that it can't release until it acquires a lock on *Table1,* the processes deadlock. When SQL Server detects a deadlock, it terminates one process's batch, rolling back the active transaction and releasing all that process's locks to resolve the deadlock. In addition to deadlocks on lock resources and memory resources, deadlocks can also occur with resources involving worker threads, parallel query execution–related resources, and MARS resources. Latches are not involved in deadlock detection because SQL Server uses deadlock-proof algorithms when it acquires latches.

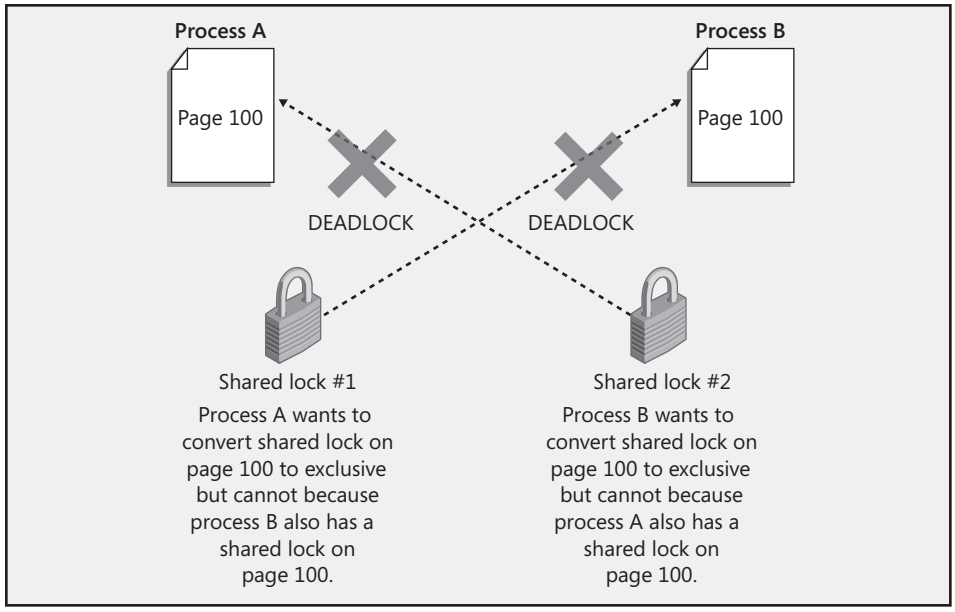


FIGURE 10-6 A conversion deadlock resulting from two processes wanting to promote their locks on the same resource within a transaction

In SQL Server, a separate thread called LOCK_MONITOR checks the system for deadlocks every five seconds. As deadlocks occur, the deadlock detection interval is reduced and can go as low as 100 milliseconds. In fact, the first few lock requests that cannot be satisfied after a deadlock has been detected will immediately trigger a deadlock search rather than wait for the next deadlock detection interval. If the deadlock frequency declines, the interval can go back to every five seconds.

This LOCK_MONITOR thread checks for deadlocks by inspecting the list of waiting locks for any cycles, which indicate a circular relationship between processes holding locks and processes waiting for locks. SQL Server attempts to choose as the victim the process that would be least expensive to roll back, considering the amount of work the process has already done. That process is killed and error message 1205 is sent to the corresponding client connection. The transaction is rolled back, meaning all its locks are released, so other processes involved in the deadlock can proceed. However, certain operations are marked as golden, or unkillable, and cannot be chosen as the deadlock victim. For example, a process involved in rolling back a transaction cannot be chosen as a deadlock victim because the changes being rolled back could be left in an indeterminate state, causing data corruption.

Using the *SET DEADLOCK_PRIORITY* statement, a process can determine its priority for being chosen as the victim if it is involved in a deadlock. There are 21 different priority levels, from –10 to 10. You can also specify the value *LOW,* which is equivalent to –5, *NORMAL*, which is equivalent to 0, and *HIGH*, which is equivalent to 5. Which session is chosen as the deadlock victim depends on each session's deadlock priority. If the sessions have different deadlock

priorities, the session with the lowest deadlock priority is chosen as the deadlock victim. If both sessions have set the same deadlock priority, SQL Server selects as the victim the session that is less expensive to roll back.

Note The lightweight latches and spinlocks used internally do not have deadlock detection services. Instead, deadlocks on latches and spinlocks are avoided rather than resolved. Avoidance is achieved via strict programming guidelines used by the SQL Server development team. These lightweight locks must be acquired in a hierarchy, and a process must not have to wait for a regular lock while holding a latch or spinlock. For example, one coding rule is that a process holding a spinlock must never directly wait for a lock or call another service that might have to wait for a lock, and a request can never be made for a spinlock that is higher in the acquisition hierarchy. By establishing similar guidelines for your development team for the order in which SQL Server objects are accessed, you can go a long way toward avoiding deadlocks in the first place.

In the example in Figure 10-5, the cycle deadlock could have been avoided if the processes had decided on a protocol beforehand—for example, if they had decided always to access the *Product* table first and the *PurchaseOrderDetail* table second. Then one of the processes gets the initial exclusive lock on the table being accessed first, and the other process waits for the lock to be released. One process waiting for a lock is normal and natural. Remember, waiting is not a deadlock.

You should always try to have a standard protocol for the order in which processes access tables. If you know that the processes might need to update the row after reading it, they should initially request an update lock, not a shared lock. If both processes request an update lock rather than a shared lock, the process that is granted an update lock is assured that the lock can later be promoted to an exclusive lock. The other process requesting an update lock has to wait. The use of an update lock serializes the requests for an exclusive lock. Other processes needing only to read the data can still get their shared locks and read. Because the holder of the update lock is guaranteed an exclusive lock, the deadlock is avoided.

In many systems, deadlocks cannot be completely avoided, but if the application handles the deadlock appropriately, the impact on any users involved, and on the rest of the system, should be minimal. (Appropriate handling implies that when error 1205 occurs, the application resubmits the batch, which most likely succeeds on the second try. Once one process is killed, its transaction is aborted, and its locks are released, the other process involved in the deadlock can finish its work and release its locks, so the environment is not conducive to another deadlock.) Although you might not be able to avoid deadlocks completely, you can minimize their occurrence. For example, you should write your applications so that your processes hold locks for a minimal amount of time; in that way, other processes won't have to wait too long for locks to be released. Although you don't usually invoke locking directly, you can influence locking by keeping transactions as short as possible. For example, don't ask for user input in the middle of a transaction. Instead, get the input first and then quickly perform the transaction.

Row Versioning

At the beginning of this chapter, I described two concurrency models that SQL Server can use. Pessimistic concurrency uses locking to guarantee the appropriate transactional behavior and avoid problems such as dirty reads, according to the isolation level you are using. Optimistic concurrency uses a new technology called *row versioning* to guarantee your transactions. Starting in SQL Server 2005, optimistic concurrency is available after you enable one or both of the database properties called READ_COMMITTED_SNAPSHOT and ALLOW_SNAPSHOT_ ISOLATION. Exclusive locks can be acquired when you use optimistic concurrency, so you still need to be aware of all issues related to lock modes, lock resources, and lock duration, as well as the resources required to keep track of and manage locks. The difference between optimistic and pessimistic concurrency is that with optimistic concurrency, writers and readers do not block each other. Or, using locking terminology, a process requesting an exclusive lock does not block when the requested resource currently has a shared lock. Conversely, a process requesting a shared lock does not block when the requested resource currently has an exclusive lock.

It is possible to avoid blocking because as soon as one of the new database options is enabled, SQL Server starts using *tempdb* to store copies (versions) of all rows that have changed, and it keeps those copies as long as there are any transactions that might need to access them. The space in *tempdb* used to store previous versions of changed rows is called the *version store*.

Overview of Row Versioning

In earlier versions of SQL Server, the tradeoff in concurrency solutions is that we can avoid having writers block readers if we are willing to risk inconsistent data—that is, if we use Read Committed isolation. If our results must always be based on committed data, we need to be willing to wait for changes to be committed.

SQL Server 2005 introduced a new isolation level called *Snapshot isolation* and a new nonblocking flavor of Read Committed isolation called *Read Committed Snapshot Isolation (RCSI)*. These row versioning–based isolation levels allow a reader to get to a previously committed value of the row without blocking, so concurrency is increased in the system. For this to work, SQL Server must keep old versions of a row when it is updated or deleted. If multiple updates are made to the same row, multiple older versions of the row might need to be maintained. Because of this, row versioning is sometimes called *multiversion concurrency control*.

To support storing multiple older versions of rows, additional disk space is used from the *tempdb* database. The disk space for the version store must be monitored and managed appropriately, and I point out some of the ways you can do that later in this section. Versioning works by making any transaction that changes data keep the old versions of the data around so that a snapshot of the database (or a part of the database) can be constructed from these old versions.

Row Versioning Details

When a row in a table or index is updated, the new row is stamped with the transaction sequence number (XSN) of the transaction that is doing the update. The XSN is a monotonically increasing number that is unique within each SQL Server database. The concept of XSN is not the same as Log Sequence Numbers (LSNs), which I discussed in Chapter 4, "Logging and Recovery." I discuss XSNs in more detail later. When updating a row, the previous version is stored in the version store, and the new row contains a pointer to the old row in the version store. Old rows in the version store might contain pointers to even older versions. All the old versions of a particular row are chained in a linked list, and SQL Server might need to follow several pointers in a list to reach the right version. Version rows must be kept in the version store only as long as there are operations that might require them.

In Figure 10-7, the current version of the row is generated by transaction T3, and it is stored in the normal data page. The previous versions of the row, generated by transaction T2 and transaction Tx, are stored in pages in the version store (in *tempdb*).

Row versioning gives SQL Server an optimistic concurrency model to work with when an application requires it or when the concurrency reduction of using the default pessimistic model is unacceptable. Before you switch to the row versioning–based isolation levels, you must carefully consider the tradeoffs of using this new concurrency model. In addition to requiring extra management to monitor the increased use of *tempdb* for the version store, versioning slows the performance of update operations due to the extra work involved in maintaining old versions. Update operations bear this cost, even if there are no current readers of the data. If there are readers using row versioning, they have the extra cost of traversing the link pointers to find the appropriate version of the requested row.

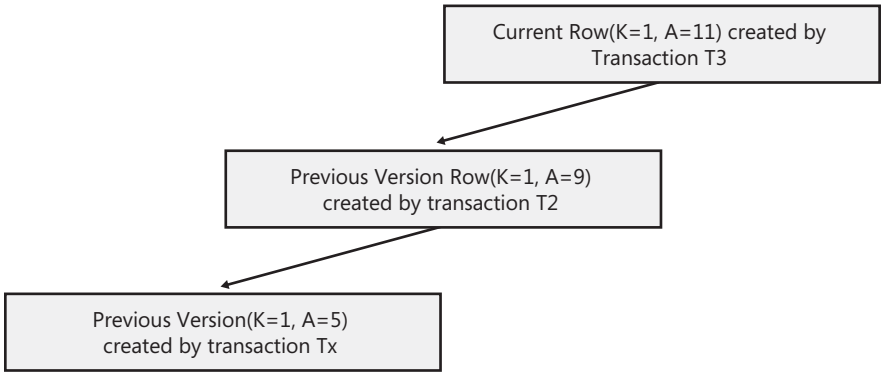


FIGURE 10-7 Versions of a row

In addition, because the optimistic concurrency model of Snapshot isolation assumes (optimistically) that not many update conflicts will occur, you should not choose the Snapshot isolation level if you are expecting contention for updating the same data concurrently. Snapshot isolation works well to enable readers not to be blocked by writers, but simultaneous writers are

still not allowed. In the default pessimistic model, the first writer will block all subsequent writers, but using Snapshot isolation, subsequent writers could actually receive error messages and the application would need to resubmit the original request. Note that these update conflicts occur only with the full Snapshot isolation, not with the enhanced RCSI.

Snapshot-Based Isolation Levels

SQL Server 2008 provides two types of snapshot-based isolation, both of which use row versioning to maintain the snapshot. One type, RCSI, is enabled simply by setting a database option. Once enabled, no further changes need to be made. Any transaction that would have operated under the default Read Committed isolation will run under RCSI. The other type, Snapshot isolation must be enabled in two places. You must first enable the database with the ALLOW_SNAPSHOT_ISOLATION option, and then each connection that wants to use SI must set the isolation level using the *SET TRANSACTION ISOLATION LEVEL* command. Let's compare these two types of Snapshot-based isolation.

Read Committed Snapshot Isolation

RCSI is a statement-level Snapshot-based isolation, which means any queries see the most recent committed values as of the beginning of the statement. For example, let's look at the scenario in Table 10-9. Assume that two transactions are running in the *AdventureWorks2008* database, which has been enabled for RCSI, and that before either transaction starts running, the *ListPrice* value of product 922 is 8.89.

TABLE 10-9 A *SELECT* Running in RCSI

Time	Transaction 1	Transaction 2

1	`BEGIN TRAN`
	`UPDATE Production.Product`
	`SET ListPrice = 10.00`
	`WHERE ProductID = 922;`
		`BEGIN TRAN`

2		`SELECT ListPrice`
		`FROM Production.Product`
		`WHERE ProductID = 922;`
		`-- SQL Server returns 8.89`

3	`COMMIT TRAN`	

4		`SELECT ListPrice`
		`FROM Production.Product`
		`WHERE ProductID = 922;`
		`-- SQL Server returns 10.00`

5		`COMMIT TRAN`

We should note that at Time = 2, the change made by Transaction 1 is still uncommitted, so the lock is still held on the row for *ProductID* = 922. However, Transaction 2 does not block on that lock; it has access to an old version of the row with a last committed *ListPrice* value

of 8.89. After Transaction 1 has committed and released its lock, Transaction 2 sees the new value of *ListPrice*. This is still Read Committed isolation (just a nonlocking variation), so there is no guarantee that read operations are repeatable.

You can consider RCSI to be just a variation of the default isolation level Read Committed. The same behaviors are allowed and disallowed, as indicated back in Table 10-2.

RCSI is enabled and disabled with the *ALTER DATABASE* command, as shown in this command to enable RCSI in the *AdventureWorks2008* database:

```
ALTER DATABASE AdventureWorks2008
   SET READ_COMMITTED_SNAPSHOT ON;
```

Ironically, although this isolation level is intended to help avoid blocking, if there are any users in the database when the preceding command is executed, the *ALTER* statement blocks. (The connection issuing the *ALTER* command can be in the database, but no other connections can be.) Until the change is successful, the database continues to operate as if it is not in RCSI mode. The blocking can be avoided by specifying a TERMINATION clause for the *ALTER* command, as discussed in Chapter 3, "Databases and Database Files":

```
ALTER DATABASE AdventureWorks2008
   SET READ_COMMITTED_SNAPSHOT ON WITH NO_WAIT;
```

If there are any users in the database, the preceding *ALTER* fails with the following error:

```
Msg 5070, Level 16, State 2, Line 1
Database state cannot be changed while other users are using
the database 'AdventureWorks2008'
Msg 5069, Level 16, State 1, Line 1
ALTER DATABASE statement failed.
```

You can also specify one of the ROLLBACK termination options, basically to break any current database connections.

The biggest benefit of RCSI is that you can introduce greater concurrency because readers do not block writers and writers do not block readers. However, writers do block writers because the normal locking behavior applies to all *UPDATE*, *DELETE*, and *INSERT* operations. No SET options are required for any session to take advantage of RCSI, so you can reduce the concurrency impact of blocking and deadlocking without any change in your applications.

Snapshot Isolation

Snapshot isolation requires using a *SET* command in the session, just like for any other change of isolation level (for example, *SET TRANSACTION ISOLATION LEVEL SERIALIZABLE*). For a session-level option to take effect, you must also allow the database to use SI by altering the database:

```
ALTER DATABASE AdventureWorks2008
   SET ALLOW_SNAPSHOT_ISOLATION ON;
```

When altering the database to allow SI, a user in the database does not necessarily block the command from completing. However, if there is an active transaction in the database, the *ALTER* is blocked. This does not mean that there is no effect until the statement completes. Changing the database to allow full SI can be a deferred operation. The database can actually be in one of four states with regard to ALLOW_SNAPSHOT_ISOLATION. It can be ON or OFF, but it can also be IN_TRANSITION_TO_ON or IN_TRANSITION_TO_OFF.

Here is what happens when you *ALTER* a database to ALLOW_SNAPSHOT_ISOLATION:

- SQL Server waits for the completion of all active transactions, and the database status is set to IN_TRANSITION_TO_ON.
- Any new *UPDATE* or *DELETE* transactions start generating versions in the version store.
- New snapshot transactions cannot start because transactions that are already in progress are not storing row versions as the data is changed. New snapshot transactions would have to have committed versions of the data to read. There is no error when you execute the *SET TRANSACTION ISOLATION LEVEL SNAPSHOT* command; the error occurs when you try to *SELECT* data, and you get this message:

  ```
  Msg 3956, Level 16, State 1, Line 1
  Snapshot isolation transaction failed to start in database 'AdventureWorks2008'
  because the ALTER DATABASE command which enables snapshot isolation for this database
  has not finished yet. The database is in transition to pending ON state. You must wait
  until the ALTER DATABASE Command completes successfully.
  ```

- As soon as all transactions that were active when the *ALTER* command began have finished, the *ALTER* can finish and the state change are complete. The database now is in the state ALLOW_SNAPSHOT_ISOLATION.

Taking the database out of ALLOW_SNAPSHOT_ISOLATION mode is similar, and again, there is a transition phase.

- SQL Server waits for the completion of all active transactions, and the database status is set to IN_TRANSITION_TO_OFF.
- New snapshot transactions cannot start.
- Existing snapshot transactions still execute snapshot scans, reading from the version store.
- New transactions continue generating versions.

Snapshot Isolation Scope

SI gives you a transactionally consistent view of the data. Any rows read are the most recent committed version of the rows as of the beginning of the transaction. (For RCSI, we get the most recent committed version as of the beginning of the statement.) A key point to keep in mind is that the transaction does not start at the *BEGIN TRAN* statement; for the purposes of SI, a transaction starts the first time the transactions accesses any data in the database.

As an example of SI, let's look at a scenario similar to the one in Table 10-9. Table 10-10 shows activities in a database with ALLOW_SNAPSHOT_ISOLATION set to ON. Assume two transactions are running in the *AdventureWorks2008* database and that before either transaction starts, the *ListPrice* value of Product 922 is 10.00.

TABLE 10-10 A *SELECT* Running in a SNAPSHOT Transaction

Time	Transaction 1	Transaction 2
1	`BEGIN TRAN`	

2	`UPDATE Production.Product`
	`SET ListPrice = 12.00`
	`WHERE ProductID = 922;`
		`SET TRANSACTION ISOLATION`
		`LEVEL SNAPSHOT`

3		`BEGIN TRAN`

4		`SELECT ListPrice`
		`FROM Production.Product`
		`WHERE ProductID = 922;`
		`-- SQL Server returns 10.00`
		`-- This is the beginning of`
		`-- the transaction`

5	`COMMIT TRAN`	

6		`SELECT ListPrice`
		`FROM Production.Product`
		`WHERE ProductID = 922;`
		`-- SQL Server returns 10.00`
		`-- Return the committed`
		`-- value as of the beginning`
		`-- of the transaction`

7		`COMMIT TRAN`

		`SELECT ListPrice`
		`FROM Production.Product`
		`WHERE ProductID = 922;`
		`-- SQL Server returns 12.00`

Even though Transaction 1 has committed, Transaction 2 continues to return the initial value it read of 10.00 until Transaction 2 completes. Only after Transaction 2 is complete does the connection read a new value for *ListPrice*.

Viewing Database State

The catalog view *sys.databases* contains several columns that report on the Snapshot isolation state of the database. A database can be enabled for SI and/or RCSI. However, enabling one does not automatically enable or disable the other. Each one has to be enabled or disabled individually using separate *ALTER DATABASE* commands.

The column *snapshot_isolation_state* has possible values of 0 to 4, indicating each of the four possible SI states, and the *snapshot_isolation_state_desc* column spells out the state. Table 10-11 summarizes what each state means.

TABLE 10-11 Possible Values for the Database Option ALLOW_SNAPSHOT_ISOLATION

Snapshot Isolation State	Description
OFF	Snapshot isolation state is disabled in the database. In other words, transactions with Snapshot isolation are not allowed. Database versioning state is initially set to OFF during recovery. If versioning is enabled, versioning state is set to ON after recovery.
IN_TRANSITION_TO_ON	The database is in the process of enabling SI. It waits for the completion of all *UPDATE* transactions that were active when the *ALTER DATABASE* command was issued. New *UPDATE* transactions in this database start paying the cost of versioning by generating row versions. Transactions using Snapshot isolation cannot start.
ON	SI is enabled. New snapshot transactions can start in this database. Existing snapshot transactions (in another snapshot-enabled session) that start before versioning state is turned ON cannot do a snapshot scan in this database because the snapshot those transactions are interested in is not properly generated by the *UPDATE* transactions.
IN_ TRANSITION_TO_OFF	The database is in the process of disabling the SI state and is unable to start new snapshot transactions. *UPDATE* transactions still pay the cost of versioning in this database. Existing snapshot transactions can still do snapshot scans. IN_TRANSITION_TO_OFF does not become OFF until all existing transactions finish.

The *is_read_committed_snapshot_on* column has a value of 0 or 1. Table 10-12 summarizes what each state means.

TABLE 10-12 Possible Values for the Database Option READ_COMMITTED_SNAPSHOT

READ_COMMITTED_SNAPSHOT State	Description
0	READ_COMMITTED_SNAPSHOT is disabled.
1	READ_COMMITTED_SNAPSHOT is enabled. Any query with Read Committed isolation executes in the nonblocking mode.

You can see the values of each of these snapshot states for all your databases with the following query:

```
SELECT name, snapshot_isolation_state_desc,
        is_read_committed_snapshot_on , *
FROM sys.databases;
```

Update Conflicts

One crucial difference between the two optimistic concurrency levels is that SI can potentially result in update conflicts when a process sees the same data for the duration of its transaction and is not blocked simply because another process is changing the same data. Table 10-13 illustrates two processes attempting to update the *Quantity* value of the same row in the *ProductInventory* table in the *AdventureWorks2008* database. Two clerks

have each received shipments of ProductID 872 and are trying to update the inventory. The *AdventureWorks2008* database has ALLOW_SNAPSHOT_ISOLATION set to ON, and before either transaction starts, the *Quantity* value of Product 872 is 324.

TABLE 10-13 An Update Conflict in SNAPSHOT Isolation

Time	Transaction 1	Transaction 2
1		SET TRANSACTION ISOLATION LEVEL SNAPSHOT
2		BEGIN TRAN
3		SELECT Quantity FROM Production.ProductInventory WHERE ProductID = 872; -- **SQL Server returns 324** -- This is the beginning of -- the transaction
4	BEGIN TRAN UPDATE Production.ProductInventory SET Quantity=Quantity + 200 WHERE ProductID = 872; -- **Quantity is now 524**	
5		UPDATE Production.ProductInventory SET Quantity=Quantity + 300 WHERE ProductID = 872; -- **Process will block**
6	COMMIT TRAN	
7		-- Process receives error 3960

The conflict happens because Transaction 2 started when the *Quantity* value was 324. When that value was updated by Transaction 1, the row version with 324 was saved in the version store. Transaction 2 continues to read that row for the duration of the transaction. If both *UPDATE* operations were allowed to succeed, we would have a classic lost update situation. Transaction 1 added 200 to the quantity, and then Transaction 2 would add 300 to the original value and save that. The 200 added by Transaction 1 would be completely lost. SQL Server does not allow that.

When Transaction 2 first tries to perform the *UPDATE*, it doesn't get an error immediately—it is simply blocked. Transaction 1 has an exclusive lock on the row, so when Transaction 2 attempts to get an exclusive lock, it is blocked. If Transaction 1 had rolled back its transaction, Transaction 2 would have been able to complete its *UPDATE*. But because Transaction 1 committed, SQL Server detects a conflict and generates the following error:

```
Msg 3960, Level 16, State 2, Line 1
Snapshot isolation transaction aborted due to update conflict. You cannot use snapshot
isolation to access table 'Production.ProductInventory' directly or indirectly in database'
AdventureWorks2008' to update, delete, or insert the row that has been modified or deleted
by another transaction. Retry the transaction or change the isolation level for the
update/delete statement.
```

Conflicts are possible only with SI because that isolation level is transaction-based, not statement-based. If the example in Table 10-13 were executed in a database using RCSI, the *UPDATE* statement executed by Transaction 2 would not use the old value of the data. It would be blocked when trying to read the current *Quantity,* and then when Transaction 1 finished, it would read the new updated *Quantity* as the current value and add 300 to that. Neither update would be lost.

If you choose to work in SI, you need to be aware that conflicts can happen. They can be minimized, but as with deadlocks, you cannot be sure that you will never have conflicts. Your application must be written to handle conflicts appropriately and not assume that the *UPDATE* has succeeded. If conflicts occur occasionally, you might consider it part of the price to be paid for using SI, but if they occur too often, you might need to take extra steps.

You might consider whether SI is really necessary, and if it is, you should determine whether the statement-based RCSI might give you the behavior you need without the cost of detecting and dealing with conflicts. Another solution is to use a query hint called UPDLOCK to make sure no other process updates data before you're ready to update it. In Table 10-13, Transaction 2 could use UPDLOCK on its initial *SELECT* as follows:

```
SELECT Quantity
FROM Production.ProductInventory WITH (UPDLOCK)
WHERE ProductID  = 872;
```

The UPDLOCK hint forces SQL Server to acquire update locks for Transaction 2 on the row that is selected. When Transaction 1 then tries to update that row, it blocks. It is not using SI, so it does not see the previous value of *Quantity*. Transaction 2 can perform its update because Transaction 1 is blocked, and it commits. Transaction 1 can then perform its update on the new value of *Quantity,* and neither update is lost.

I will provide a few more details about locking hints at the end of this chapter.

Data Definition Language and SNAPSHOT Isolation

When working with SI, you need to be aware that although SQL Server keeps versions of all the changed data, that metadata is not versioned. Therefore, certain DDL statements are not allowed inside a snapshot transaction. The following DDL statements are disallowed in a snapshot transaction:

- *CREATE / ALTER / DROP INDEX*
- *DBCC DBREINDEX*
- *ALTER TABLE*
- *ALTER PARTITION FUNCTION / SCHEME*

On the other hand, the following DDL statements are allowed:

- *CREATE TABLE*
- *CREATE TYPE*
- *CREATE PROC*

Note that the allowable DDL statements are ones that create brand-new objects. In SI, there is no chance that any simultaneous data modifications affect the creation of these objects. Table 10-14 shows a pseudo-code example of a snapshot transaction that includes both *CREATE TABLE* and *CREATE INDEX*.

TABLE 10-14 DDL Inside a SNAPSHOT Transaction

Time	Transaction 1	Transaction 2
1	`SET TRANSACTION ISOLATION LEVEL SNAPSHOT;`	
2	`BEGIN TRAN`	
3	`SELECT count(*) FROM Production.Product; -- This is the beginning of -- the transaction`	
4		`BEGIN TRAN`
5	`CREATE TABLE NewProducts ( <column definitions>) -- This DDL is legal`	`INSERT Production.Product VALUES (9999, .....) -- A new row is insert into -- the Product table`
6		`COMMIT TRAN`
7	`CREATE INDEX PriceIndex ON Production.Product (ListPrice) -- This DDL will generate an -- error`	

The *CREATE TABLE* statement succeeds even though Transaction 1 is in SI because it is not affected by anything any other process can do. The *CREATE INDEX* statement is a different story. When Transaction 1 started, the new row with ProductID 9999 did not exist. But when the *CREATE INDEX* statement is encountered, the *INSERT* from Transaction 2 has been committed. Should Transaction 1 include the new row in the index? There is actually no way to avoid including the new row, but that would violate the snapshot that Transaction 1 is using, and SQL Server generates an error instead of creating the index.

Another aspect of concurrent DDL to consider is what happens when a statement outside the snapshot transaction changes an object referenced by a snapshot transaction. The DDL is allowed, but you can get an error in the snapshot transaction when this happens. Table 10-15 shows an example.

TABLE 10-15 Concurrent DDL Outside the SNAPSHOT Transaction

Time	Transaction 1	Transaction 2
1	`SET TRANSACTION ISOLATION LEVEL SNAPSHOT;`	
2	`BEGIN TRAN`	
3	`SELECT TOP 10 * FROM Production.Product; -- This is the start of -- the transaction`	
4		`BEGIN TRAN ALTER TABLE Purchasing.Vendor ADD notes varchar(1000); COMMIT TRAN`
5	`SELECT TOP 10 * FROM Production.Product; -- Succeeds -- The ALTER to a different -- table does not affect -- this transaction`	
6		`BEGIN TRAN ALTER TABLE Production.Product ADD LowestPrice money; COMMIT TRAN`
7	`SELECT TOP 10 * FROM Production. Product; -- ERROR`	

For the preceding situation, in Transaction 1, the repeated *SELECT* statements should always return the same data from the *Product* table. An external *ALTER TABLE* on a completely different table has no effect on the snapshot transaction, but Transaction 2 then alters the *Product* table to add a new column. Because the metadata representing the former table structure is not versioned, Transaction 1 cannot produce the same results for the third *SELECT*. SQL Server generates this error:

```
Msg 3961, Level 16, State 1, Line 1
Snapshot isolation transaction failed in database 'AdventureWorks2008' because the object
accessed by the statement has been modified by a DDL statement in another concurrent
transaction since the start of this transaction. It is disallowed because the metadata is
not versioned. A concurrent update to metadata can lead to inconsistency if mixed with
snapshot isolation.
```

In this version, any concurrent change to metadata on objects referenced by a snapshot transaction generates this error, even if there is no possibility of anomalies. For example, if Transaction 1 issues a *SELECT count(*),* which is not affected by the *ALTER TABLE* statement, SQL Server still generates error 3961.

Summary of Snapshot-Based Isolation Levels

SI and RCSI are similar in the sense that they are based on the versioning of rows in a database. However, there are some key differences in how these options are enabled from an administration perspective and also in how they affect your applications. I have discussed many of these differences already, but for completeness, Table 10-16 lists both the similarities and the differences between the two types of snapshot-based isolation.

TABLE 10-16 Snapshot vs. Read Committed Snapshot Isolation

Snapshot Isolation	Read Committed Snapshot Isolation
The database must be configured to allow SI, and the session must issue the command *SET TRANSACTION ISOLATION LEVEL SNAPSHOT*.	The database must be configured to use RCSI, and sessions must use the default isolation level. No code changes are required.
Enabling SI for a database is an online operation. It allows a DBA to turn on versioning for one particular application such as one that is creating large reports. The DBA can then turn off versioning after the reporting transaction has started to prevent new snapshot transactions from starting. Turning on SI in an existing database is synchronous. When the *ALTER DATABASE* command is given, control does not return to the DBA until all existing update transactions that need to create versions in the current database finish. At this time, ALLOW_SNAPSHOT_ISOLATION is changed to ON. Only then can users start a snapshot transaction in that database. Turning off SI is also synchronous.	Enabling RCSI for a database requires a SHARED_TRANSACTION_WORKSPACE lock on the database. All users must be kicked out of a database to enable this option.
There are no restrictions on active sessions in the database when this database option is enabled.	There should be no other sessions active in the database when you enable this option.
If an application runs a snapshot transaction that accesses tables from two databases, the DBA must turn on ALLOW_SNAPSHOT_ISOLATION in both databases before the application starts a snapshot transaction.	RCSI is really a table-level option, so tables from two different databases, referenced in the same query, can each have their own individual setting. One table might get its data from the version store, while the other table is reading only the current versions of the data. There is no requirement that both databases must have the RCSI option enabled.
The IN_TRANSITION versioning states do not persist. Only the ON and OFF states are remembered on disk.	There are no IN_TRANSITION states here. Only ON and OFF states persist.

TABLE 10-16 Snapshot vs. Read Committed Snapshot Isolation

Snapshot Isolation	Read Committed Snapshot Isolation
When a database is recovered after a server crash, or after your SQL Server instance is shut down, restored, attached, or made ONLINE, all versioning history for that database is lost. If database versioning state is ON, SQL Server can allow new snapshot transactions to access the database, but must prevent previous snapshot transactions from accessing the database. Those previous transactions would need to access data from a point in time before the database recovers.	This is an object-level option; it is not at the transaction level, so it is not applicable.
If the database is in the IN_TRANSITION_ TO_ON state, *ALTER DATABASE SET ALLOW_ SNAPSHOT_ ISOLATION OFF* waits for about six seconds and might fail if the database state is still in the IN_TRANSITION_TO_ON state. The DBA can retry the command after the database state changes to ON.	This option can be enabled only when there is no other active session in the database, so there are no transitional states.
For read-only databases, versioning is automatically enabled. You still can use *ALTER DATABASE SET ALLOW_SNAPSHOT_ISOLATION ON* for a read-only database. If the database is made read-write later, versioning for the database is still enabled.	As for SI, versioning is enabled automatically for read-only databases.
If there are long-running transactions, a DBA might need to wait a long time before the versioning state change can finish. A DBA can cancel the wait, and the versioning state is rolled back and set to the previous one.	This option can be enabled only when there is no other active session in the database, so there are no transitional states.
You can change the versioning state of *tempdb*. The versioning state of *tempdb* is preserved when SQL Server restarts, although the content of *tempdb* is not preserved.	You cannot turn this option ON for *tempdb*.
You can change the versioning state of the *master* database.	You cannot change this option for the *master* database.
You can change the versioning state of model. If versioning is enabled for model, every new database created will have versioning enabled as well. However, the versioning state of *tempdb* is not automatically enabled if you enable versioning for *model*.	Similar to the behavior for SI, except that there are no implications for *tempdb*.

TABLE 10-16 Snapshot vs. Read Committed Snapshot Isolation

Snapshot Isolation	Read Committed Snapshot Isolation
You can turn this option ON for *msdb*.	You cannot turn on this option ON for *msdb* because this can potentially break the applications built on *msdb* that rely on blocking behavior of Read Committed isolation.
A query in a SI transaction sees data that was committed before the start of the transaction, and each statement in the transaction sees the same set of committed changes.	A statement running in RCSI sees everything committed before the start of the statement. Each new statement in the transaction picks up the most recent committed changes.
SI can result in update conflicts that might cause a rollback or abort the transaction.	There is no possibility of update conflicts.

The Version Store

As soon as a database is enabled for ALLOW_SNAPSHOT_ISOLATION or READ_COMMITTED_SNAPSHOT, all *UPDATE* and *DELETE* operations start generating row versions of the previously committed rows, and they store those versions in the version store on data pages in *tempdb*. Version rows must be kept in the version store only so long as there are snapshot queries that might need them.

SQL Server 2008 provides several DMVs that contain information about active snapshot transactions and the version store. We won't examine all the details of all those DMVs, but we look at some of the crucial ones to help you determine how much use is being made of your version store and what snapshot transactions might be affecting your results. The first DMV we look at, *sys.dm_tran_version_store*, contains information about the actual rows in the version store. Run the following script to make a copy of the *Production.Product* table, and then turn on ALLOW_SNAPSHOT_ISOLATION in the *AdventureWorks2008* database. Finally, verify that the option is ON and that there are currently no rows in the version store. You might need to close any active transactions currently using *AdventureWorks2008*:

```
USE AdventureWorks2008
SELECT * INTO NewProduct
FROM Production.Product;
GO
ALTER DATABASE ADVENTUREWORKS2008 SET ALLOW_SNAPSHOT_ISOLATION ON;
GO
SELECT name, snapshot_isolation_state_desc,
      is_read_committed_snapshot_on
FROM sys.databases
WHERE name= AdventureWorks2008;
GO
SELECT COUNT(*) FROM sys.dm_tran_version_store;
GO
```

As soon as you see that the database option is ON and there are no rows in the version store, you can continue. What I want to illustrate is that as soon as ALLOW_SNAPSHOT_ ISOLATION

is enabled, SQL Server starts storing row versions, even if no snapshot transactions need to read those versions. So now run this *UPDATE* statement on the *NewProduct* table and look at the version store again:

```
UPDATE  NewProduct
SET ListPrice = ListPrice * 1.1;
GO
SELECT COUNT(*) FROM sys.dm_tran_version_store;
GO
```

You should see that there are now 504 rows in the version store because there are 504 rows in the *NewProduct* table. The previous version of each row, prior to the update, has been written to the version store in *tempdb*.

Note SQL Server starts generating versions in *tempdb* as soon as a database is enabled for one of the snapshot-based isolation levels. In a heavily updated database, this can affect the behavior of other queries that use *tempdb*, as well as the server itself.

As shown earlier in Figure 10-7, the version store maintains link lists of rows. The current row points to the next older row, which can point to an older row, and so on. The end of the list is the oldest version of that particular row. To support row versioning, a row needs 14 additional bytes of information to keep track of the pointers. Eight bytes are needed for the actual pointer to the file, page, and row in *tempdb*, and 6 bytes are needed to store the XSN to help SQL Server determine which rows are current, or which versioned row is the one that a particular transaction needs to access. I tell you more about the XSN when we look at some of the other snapshot transaction metadata. In addition, one of the bits in the first byte of each data row (the TagA byte) is turned on to indicate that this row has versioning information in it.

Any row inserted or updated when a database is using one of the snapshot-based isolation levels will contain these 14 extra bytes. The following code creates a small table and inserts two rows into it in the *AdventureWorks2008* database, which already has ALLOW_SNAPSHOT_ISOLATION enabled. I then find the page number using *DBCC IND* (it is page 6,709) and use DBCC to look at the rows on the page. The output shows only one of the rows inserted:

```
CREATE TABLE T1 (T1ID char(1), T1name char(10));
GO
INSERT T1 SELECT 'A', 'aaaaaaaaaa';
INSERT T1 SELECT 'B', 'bbbbbbbbbb';
GO
DBCC IND (AdventureWorks2008, 'T1',-1); -- page 6709
DBCC TRACEON (3604);
DBCC PAGE('AdventureWorks2008', 1, 6709, 1);
OUTPUT ROW:
Slot 0, Offset 0x60, Length 32, DumpStyle BYTE
Record Type = PRIMARY_RECORD
Record Attributes =  NULL_BITMAP VERSIONING_INFO
```

```
Memory Dump @0x6207C060
00000000:   50000f00 41616161 61616161 61616102 †P...Aaaaaaaaaaa.
00000010:   00fc0000 00000000 0000020d 00000000 †................
```

I have highlighted the new header information that indicates this row contains versioning information, and I have also highlighted the 14 bytes of the versioning information. The XSN is all 0's in the row because it was not modified as part of a transaction that Snapshot isolation needs to keep track of. *INSERT* statements create new data that no snapshot transaction needs to see. If I update one of these rows, the previous row is written to the version store and the XSN is reflected in the row versioning information:

```
UPDATE T1 SET T1name = '2222222222' where T1ID = 'A';
GO
DBCC PAGE('AdventureWorks2008', 1, 6709, 1);
GO
OUTPUT ROW:
Slot 0, Offset 0x60, Length 32, DumpStyle BYTE
Record Type = PRIMARY_RECORD
Record Attributes =  NULL_BITMAP VERSIONING_INFO
Memory Dump @0x61C4C060
00000000:   50000f00 41323232 32323232 32323202 †P...A2222222222.
00000010:   00fc1804 00000100 0100590d 00000000 †..........Y.....
```

As mentioned, if your database is enabled for one of the snapshot-based isolation levels, every new row has an additional 14 bytes added to it whether or not that row is ever actually involved in versioning. Every row updated also has the 14 bytes added to it, if they aren't already part of the row, and the update is done as a *DELETE* followed by an *INSERT*. This means that for tables and indexes on full pages, a simple *UPDATE* could result in page splitting.

When a row is deleted in a database enabled for snapshots, a pointer is left on the page as a ghost record to point to the deleted row in the version store. These ghost records are very similar to the ones we saw in Chapter 6, "Indexes: Internals and Management," and they're cleaned up as part of the versioning cleanup process, as I discuss shortly. Here's an example of a ghost record under versioning:

```
DELETE T1 WHERE T1ID = 'B';
DBCC PAGE('AdventureWorks2008 ', 1, 6709, 1);
GO
--Partial Results:
Slot 4, Offset 0x153, Length 15, DumpStyle BYTE

Record Type = GHOST_VERSION_RECORD
Record Attributes =  VERSIONING_INFO
Memory Dump @0x5C0FC153

00000000:   4ef80300 00010000 00210200 000000††††N........!.....
```

The record header indicates that this row is a GHOST_VERSION_RECORD and that it contains versioning information. The actual data, however, is not on the row, but the XSN is, so that snapshot transactions know when this row was deleted and whether they should access

the older version of it in their snapshot. The *sys.dm_db_index_physical_stats* DMV that was discussed in Chapter 6 contains the count of ghost records due to versioning (*version_ghost_record_count*) and the count of all ghost records (*ghost_record_count*), which includes the versioning ghosts. If an update is performed as a *DELETE* followed by an *INSERT* (not in place), both the ghost for the old value and the new value must exist simultaneously, increasing the space requirements for the object.

If a database is in a snapshot-based isolation level, all changes to both data and index rows must be versioned. A snapshot query traversing an index still needs access to index rows pointing to the older (versioned) rows. So in the index levels, we might have old values, as ghosts, existing simultaneously with the new value, and the indexes can require more storage space.

The extra 14 bytes of versioning information can be removed if the database is changed to a non-snapshot isolation level. Once the database option is changed, each time a row containing versioning information is updated, the versioning bytes are removed.

Management of the Version Store

The version store size is managed automatically, and SQL Server maintains a cleanup thread to make sure versioned rows are not kept around longer than needed. For queries running under SI, the row versions must be kept until the end of the transaction. For *SELECT* statements running under RCSI, a particular row version is not needed once the *SELECT* statement has executed and it can be removed.

The regular cleanup function is performed every minute as a background process to reclaim all reusable space from the version store. If *tempdb* actually runs out of free space, the cleanup function is called before SQL Server increases the size of the files. If the disk gets so full that the files cannot grow, SQL Server stops generating versions. If that happens, a snapshot query fails if it needs to read a version that was not generated due to space constraints. Although a full discussion of troubleshooting and monitoring is beyond the scope of this book, I will point out that SQL Server 2008 includes more than a dozen performance counters to monitor *tempdb* and the version store. These include counters to keep track of transactions that use row versioning. The following counters are contained in the SQLServer:Transactions performance object. Additional details and additional counters can be found in *SQL Server Books Online*.

- **Free Space in *tempdb*** This counter monitors the amount of free space in the *tempdb* database. You can observe this value to detect when *tempdb* is running out of space, which might lead to problems keeping all the necessary version rows.
- **Version Store Size** This counter monitors the size in kilobytes of the version store. Monitoring this counter can help determine a useful estimate of the additional space you might need for *tempdb*.
- **Version Generation Rate and Version Cleanup Rate** These counters monitor the rate at which space is acquired and released from the version store, in kilobytes per second.

- **Update Conflict Ratio** This counter monitors the ratio of update snapshot transactions that have update conflicts. It is the ratio of the number of conflicts compared to the total number of update snapshot transactions.
- **Longest Transaction Running Time** This counter monitors the longest running time in seconds of any transaction using row versioning. It can be used to determine whether any transaction is running for an unreasonable amount of time, as well as help you determine the maximum size needed in *tempdb* for the version store.
- **Snapshot Transactions** This counter monitors the total number of active snapshot transactions.

Snapshot Transaction Metadata

The most important DMVs for observing snapshot transaction behavior are *sys.dm_tran_version_store* (which we briefly looked at earlier in this chapter), *sys.dm_tran_transactions_snapshot,* and *sys.dm_tran_active_snapshot_database_transactions*.

All these views contain a column called *transaction_sequence_num,* which is the XSN that I mentioned earlier. Each transaction is assigned a monotonically increasing XSN value when it starts a snapshot read or when it writes data in a snapshot-enabled database. The XSN is reset to 0 when your SQL Server instance is restarted. Transactions that do not generate version rows and do not use snapshot scans do not receive an XSN.

Another column, *transaction_id,* is also used in some of the snapshot transaction metadata. A transaction ID is a unique identification number assigned to the transaction. It is used primarily to identify the transaction in locking operations. It can also help you identify which transactions are involved in snapshot operations. The transaction ID value is incremented for every transaction across the whole server, including internal system transactions, so whether or not that transaction is involved in any snapshot operations, the current transaction ID value is usually much larger than the current XSN.

You can check current transaction number information using the view *sys.dm_tran_current_transaction*, which returns a single row containing the following columns:

- ***transaction_id*** This value displays the transaction ID of the current transaction. If you are selecting from the view inside a user-defined transaction, you should continue to see the same *transaction_id* every time you select from the view. If you are running a *SELECT* from *sys.dm_tran_current_transaction* outside of transaction, the *SELECT* itself generates a new *transaction_id* value and you see a different value every time you execute the same *SELECT*, even in the same connection.
- ***transaction_sequence_num*** This value is the XSN of the current transaction, if it has one. Otherwise, this column returns 0.
- ***transaction_is_snapshot*** This value is 1 if the current transaction was started under SNAPSHOT isolation; otherwise, it is 0. (That is, this column is 1 if the current session has set TRANSACTION ISOLATION LEVEL to SNAPSHOT explicitly.)

- ***first_snapshot_sequence_num*** When the current transaction started, it took a snapshot of all active transactions, and this value is the lowest XSN of the transactions in the snapshot.
- ***last_transaction_sequence_num*** This value is the most recent XSN generated by the system.
- ***first_useful_sequence_num*** This value is an XSN representing the upper bound of version store rows that can be cleaned up without affecting any transactions. Any rows with an XSN less than this value are no longer needed.

I now create a simple versioning scenario to illustrate how the values in the snapshot metadata get updated. This is not a complete overview, but it should get you started in exploring the versioning metadata for your own queries. I use the *AdventureWorks2008* database, which has ALLOW_SNAPSHOT_ISOLATION set to ON, and I create a simple table:

```
CREATE TABLE t1
(col1 int primary key, col2 int);
GO
INSERT INTO t1 SELECT 1,10;
INSERT INTO t1 SELECT 2,20;
INSERT INTO t1 SELECT 3,30;
```

We call this session Connection 1. Change the session's isolation level, start a snapshot transaction, and examine some of the metadata:

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
GO
BEGIN TRAN
SELECT * FROM t1;
GO
select * from sys.dm_tran_current_transaction;
select * from sys.dm_tran_version_store;
select * from sys.dm_tran_transactions_snapshot;
```

The *sys.dm_tran_current_transaction* view should show you something like this: the current transaction does have an XSN, and the transaction is a snapshot transaction. Also, you can note that the *first_useful_sequence_num* value is the same as this transaction's XSN because no other snapshot transactions are valid now. I refer to this transaction's XSN as XSN1.

The version store should be empty (unless you've done other snapshot tests within the last minute). Also, *sys.dm_tran_transactions_snapshot* should be empty, indicating that there were no snapshot transactions that started when other transactions were in process.

In another connection (Connection 2), run an update and examine some of the metadata for the current transaction:

```
BEGIN TRAN
 UPDATE T1 SET col2 = 100
   WHERE col1 = 1;
SELECT * FROM sys.dm_tran_current_transaction;
```

Note that although this transaction has an XSN because it generates versions, it is not running in SI, so the *transaction_is_snapshot* value is 0. I refer to this transaction's XSN as XSN2.

Now start a third transaction in a Connection 3 to perform another *SELECT*. (Don't worry, this is the last one and we won't be keeping it around.) It is almost identical to the first, but there is an important difference in the metadata results:

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
GO
BEGIN TRAN
SELECT * FROM t1;
GO
select * from sys.dm_tran_current_transaction;
select * from sys.dm_tran_transactions_snapshot;
```

In the *sys.dm_tran_current_transaction* view, you see a new XSN for this transaction (XSN3), and you see that the value for *first_snapshot_sequence_num* and *first_useful_sequence_num* are both the same as XSN1. In the *sys.dm_tran_transactions_snapshot* view, you see that this transaction with XSN3 has two rows, indicating the two transactions that were active when this one started. Both XSN1 and XSN2 show up in the *snapshot_sequence_num* column. You can now either commit or roll back this transaction, and then close the connection.

Go back to Connection 2, where you started the *UPDATE*, and commit the transaction.

Now let's go back to the first *SELECT* transaction in Connection 1 and rerun the *SELECT* statement, staying in the same transaction:

```
SELECT * FROM t1;
```

Even though the *UPDATE* in Connection 2 has committed, we still see the original data values because we are running a snapshot transaction. We can examine the *sys.dm_tran_active_snapshot_database_transactions* view with this query:

```
SELECT transaction_sequence_num, commit_sequence_num,
   is_snapshot, session_id,first_snapshot_sequence_num,
   max_version_chain_traversed, elapsed_time_seconds
FROM  sys.dm_tran_active_snapshot_database_transactions;
```

I am not showing you the output here because it is too wide for the page, but there are many columns that you should find interesting. In particular, the *transaction_sequence_num* column contains XSN1, which is the XSN for the current connection. You could actually run the preceding query from any connection; it shows *all* active snapshot transactions in the SQL Server instance, and because it includes the *session_id,* you can join it to *sys.dm_exec_sessions* to get information about the connection that is running the transaction:

```
SELECT transaction_sequence_num, commit_sequence_num,
    is_snapshot, t.session_id,first_snapshot_sequence_num,
    max_version_chain_traversed, elapsed_time_seconds,
    host_name, login_name, transaction_isolation_level
```

```
FROM  sys.dm_tran_active_snapshot_database_transactions t
   JOIN sys.dm_exec_sessions s
     ON t.session_id = s.session_id;
```

Another value to note is in the column called *max_version_chain_traversed.* Although now it should be 1, we can change that. Go back to Connection 2 and run another *UPDATE* statement. Even though the *BEGIN TRAN* and *COMMIT TRAN* are not necessary for a single statement transaction, I am including them to make it clear that this transaction is complete:

```
BEGIN TRAN
 UPDATE T1 SET col2 = 300
   WHERE col1 = 1;
COMMIT TRAN;
```

Examine the version store if desired, to see rows being added:

```
SELECT *
 FROM sys.dm_tran_version_store;
```

When you go back to Connection 1 and run the same *SELECT* inside the original transaction and look again at the *max_version_chain_traversed* column in *sys.dm_tran_active_snapshot_database_transactions*, you should see that the number keeps growing. Repeated *UPDATE* operations, either in Connection 2 or a new connection, cause the *max_version_chain_traversed* value to just keep increasing, as long as Connection 1 stays in the same transaction. Keep this in mind as an added cost of using Snapshot isolation. As you perform more updates on data needed by snapshot transactions, your read operations take longer because SQL Server must traverse a longer version chain to get the data needed by your transactions.

This is just the tip of the iceberg regarding how the snapshot and transaction metadata can be used to examine the behavior of your snapshot transactions.

Choosing a Concurrency Model

Pessimistic concurrency is the default in SQL Server 2008 and was the only choice in all versions of SQL Server prior to SQL Server 2005. Transactional behavior is guaranteed by locking, at the cost of greater blocking. When accessing the same data resources, readers can block writers and writers can block readers. Because SQL Server was initially designed and built to use pessimistic concurrency, you should consider using that model unless you can verify that optimistic concurrency really will work better for you and your applications. If you find that the cost of blocking is becoming excessive you can consider using optimistic concurrency.

In most situations, RCSI is recommended over Snapshot isolation for several reasons:

- RCSI consumes less *tempdb* space than SI.
- RCSI works with distributed transactions; SI does not.
- RCSI does not produce update conflicts.

- RCSI does not require any change in your applications. All that is needed is one change to the database options. Any of your applications written using the default Read Committed isolation level automatically uses RCSI after making the change at the database level.

You can consider using SI in the following situations:

- The probability is low that any of your transactions have to be rolled back because of an update conflict.
- You have reports that need to be generated based on long-running, multistatement queries that must have point-in-time consistency. Snapshot isolation provides the benefit of repeatable reads without being blocked by concurrent modification operations.

Optimistic concurrency does have benefits, but you must also be aware of the costs. To summarize the benefits:

- *SELECT* operations do not acquire shared locks, so readers and writers do not block each other.
- All *SELECT* operations retrieve a consistent snapshot of the data.
- The total number of locks needed is greatly reduced compared to pessimistic concurrency, so less system overhead is used.
- SQL Server needs to perform fewer lock escalations.
- Deadlocks are less likely to occur.

Now let's summarize the other side. When weighing your concurrency options, you must consider the cost of the snapshot-based isolation levels:

- *SELECT* performance can be affected negatively when long-version chains must be scanned. The older the snapshot, the more time it takes to access the required row in an SI transaction.
- Row versioning requires additional resources in *tempdb*.
- Whenever either of the snapshot-based isolation levels are enabled for a database, *UPDATE* and *DELETE* operations must generate row versions. (Although I mentioned earlier that *INSERT* operations do not generate row versions, there are some cases where they might. In particular, if you insert a row into a table with a unique index, if there was an older version of the row with the same key value as the new row and that old row still exists as a ghost, your new row generates a version.)
- Row versioning information increases the size of every affected row by 14 bytes.
- *UPDATE* performance might be slower due to the work involved in maintaining the row versions.

- *UPDATE* operations using SI might have to be rolled back because of conflict detection. Your applications must be programmed to deal with any conflicts that occur.
- The space in *tempdb* must be carefully managed. If there are very long-running transactions, all the versions generated by update transactions during the time must be kept in *tempdb*. If *tempdb* runs out of space, *UPDATE* operations won't fail, but *SELECT* operations that need to read versioned data might fail.

To maintain a production system using SI, you should allocate enough disk space for *tempdb* so that there is always at least 10 percent free space. If the free space falls below this threshold, system performance might suffer because SQL Server expends more resources trying to reclaim space in the version store. The following formula can give you a rough estimate of the size required by version store. For long-running transactions, it might be useful to monitor the generation and cleanup rate using Performance Monitor, to estimate the maximum size needed:

```
[size of common version store] =
2 * [version store data generated per minute]
* [longest running time (minutes) of the transaction]
```

Controlling Locking

The SQL Server Query Optimizer usually chooses the correct type of lock and the lock mode. You should override this behavior only if thorough testing has shown that a different approach is preferable. Keep in mind that by setting an isolation level, you have an impact on the locks that are held, the conflicts that cause blocking, and the duration of your locks. Your isolation level is in effect for an entire session, and you should choose the one that provides the data consistency required by your application. Table-level locking hints can be used to change the default locking behavior only when necessary. Disallowing a locking level can adversely affect concurrency.

Lock Hints

T-SQL syntax allows you to specify locking hints for individual tables when they are referenced in *SELECT*, *INSERT*, *UPDATE*, and *DELETE* statements. The hints tell SQL Server the type of locking or row versioning to use for a particular table in a particular query. Because these hints are specified in a FROM clause, they are called *table-level hints*. *SQL Server Books Online* lists other table-level hints besides locking hints, but the vast majority of them affect locking behavior. They should be used only when you absolutely need finer control over locking at the object level than what is provided by your session's isolation level. The SQL Server locking hints can override the current transaction isolation level for the session. In this section, I will mention only some of the locking hints that you might need to obtain the desired concurrency behavior.

Many of the locking hints work only in the context of a transaction. However, every *INSERT*, *UPDATE*, and *DELETE* statement is automatically in a transaction, so the only concern is when you use a locking hint with a *SELECT* statement. To get the benefit of most of the following hints when used in a *SELECT* query, you must use an explicit transaction, starting with *BEGIN TRAN* and terminating with either *COMMIT TRAN* or *ROLLBACK TRAN*. The lock hint syntax is as follows:

```
SELECT select_list
FROM object [WITH (locking hint)]

DELETE [FROM] object [WITH (locking hint)]
[WHERE <search conditions>]

UDPATE object [WITH (locking hint)]
SET <set_clause>
[WHERE <search conditions>]

INSERT [INTO] object [WITH (locking hint)]
<insert specification>
```

Tip Not all the locking hints require the keyword *WITH*, but the syntax without *WITH* will go away in a future version of SQL Server. It is recommended that all hints be specified using *WITH*.

You can specify one of the following keywords for the locking hint:

- **HOLDLOCK** This hint is equivalent to the SERIALIZABLE hint. Using this hint is similar to specifying *SET TRANSACTION ISOLATION LEVEL SERIALIZABLE*, except that the SET option affects all tables, not only the one specified in this hint.
- **UPDLOCK** This hint forces SQL Server to take update locks instead of shared locks while reading the table and holds them until the end of the transaction. Taking update locks can be an important technique for eliminating conversion deadlocks.
- **TABLOCK** This hint forces SQL Server to take a shared lock on the table even if page locks would be taken otherwise. This hint is useful when you know you escalate to a table lock or if you need to get a complete snapshot of a table. You can use this hint with HOLDLOCK if you want the table lock held until the end of the transaction block to operate in Repeatable Read isolation. If you use this hint with a *DELETE* statement on a heap, it allows SQL Server to deallocate the pages as the rows are deleted. (If row or page locks are obtained when deleting from a heap, space will not be deallocated and cannot be reused by other objects.)
- **PAGLOCK** This hint forces SQL Server to take shared page locks when a single shared table lock might otherwise be taken. (To request an exclusive page lock, you must use the XLOCK hint along with the PAGLOCK hint.)
- **TABLOCKX** This hint forces SQL Server to take an exclusive lock on the table that is held until the end of the transaction block. (All exclusive locks are held until the end of

a transaction, regardless of the isolation level in effect. This hint has the same effect as specifying both the TABLOCK and the XLOCK hints together.)

- **ROWLOCK** This hint specifies that a shared row lock should be taken when a single shared page or table lock is normally taken.
- **READUNCOMMITTED | REPEATABLEREAD | SERIALIZABLE** These hints specify that SQL Server should use the same locking mechanisms as when the transaction isolation level is set to the level of the same name. However, the hint controls locking for a single table in a single statement, as opposed to locking all tables in all statements in a transaction.
- **READCOMMITTED** This hint specifies that *SELECT* operations comply with the rules for the Read Committed isolation level by using either locking or row versioning. If the database option READ_COMMITTED_SNAPSHOT is OFF, SQL Server uses shared locks and releases them as soon as the read operation is completed. If the database option READ_COMMITTED_SNAPSHOT is ON, SQL Server does not acquire locks and uses row versioning.
- **READCOMMITTEDLOCK** This hint specifies that *SELECT* statements use the locking version of Read Committed isolation (the SQL Server default). No matter what the setting is for the database option READ_COMMITTED_SNAPSHOT, SQL Server acquires shared locks when it reads the data and releases those locks when the read operation is completed.
- **NOLOCK** This hint allows uncommitted, or dirty, reads. Shared locks are not requested so that the statement does not block when reading data that is holding exclusive locks. In other words, no locking conflict is detected. This hint is equivalent to READUNCOMMITTED.
- **READPAST** This hint specifies that locked rows are skipped (read past). READPAST applies only to transactions operating at the READ COMMITTED isolation level and reads past row-level locks only.
- **XLOCK** This hint specifies that SQL Server should take an exclusive lock that is held until the end of the transaction on all data processed by the statement. This lock can be specified with either PAGLOCK or TABLOCK, in which case the exclusive lock applies to the specified resource.

Setting a Lock Timeout

Setting a LOCK_TIMEOUT also lets you control SQL Server locking behavior. By default, SQL Server does not time out when waiting for a lock; it assumes optimistically that the lock will be released eventually. Most client programming interfaces allow you to set a general timeout limit for the connection so a query is canceled by the client automatically if no response comes back after a specified amount of time. However, the message that comes back when the time period is exceeded does not indicate the cause of the cancellation; it could be because of a lock not being released, it could be because of a slow network, or it could just be a long-running query.

Like other SET options, SET LOCK_TIMEOUT is valid only for your current connection. Its value is expressed in milliseconds and can be accessed by using the system function @@LOCK_TIMEOUT. This example sets the LOCK_TIMEOUT value to five seconds and then retrieves that value for display:

```
SET LOCK_TIMEOUT 5000;
SELECT @@LOCK_TIMEOUT;
```

If your connection exceeds the lock timeout value, you receive the following error message:

```
Server: Msg 1222, Level 16, State 50, Line 1
Lock request time out period exceeded.
```

Setting the LOCK_TIMEOUT value to 0 means that SQL Server does not wait at all for locks. It basically cancels the entire statement and goes on to the next one in the batch. This is not the same as the READPAST hint, which skips individual rows.

The following example illustrates the difference between READPAST, READUNCOMMITTED, and setting LOCK_TIMEOUT to 0. All these techniques let you avoid blocking problems, but the behavior is slightly different in each case.

1. In a new query window, execute the following batch to lock one row in the *HumanResources.Department* table:

   ```
   USE AdventureWorks2008;
   BEGIN TRAN;
   UPDATE HumanResources.Department
   SET ModifiedDate = getdate()
   WHERE DepartmentID = 1;
   ```

2. Open a second connection, and execute the following statements:

   ```
   USE AdventureWorks2008;
   SET LOCK_TIMEOUT 0;
   SELECT * FROM HumanResources.Department;
   SELECT * FROM Sales.SalesPerson;
   ```

   Notice that after error 1222 is received, the second *SELECT* statement is executed, returning all 17 rows from the *SalesPerson* table. The batch is not cancelled when error 1222 is encountered.

Warning Not only is a batch not cancelled when a lock timeout error is encountered, but any active transaction will not be rolled back. If you have two *UPDATE* statements in a transaction and both must succeed if either succeeds, a lock timeout for one of the *UPDATE* statements will still allow the other statement to be processed. You must include error handling in your batch to take appropriate action in the event of an error 1222.

3. Open a third connection, and execute the following statements:

```
USE AdventureWorks2008 ;
SELECT * FROM HumanResources.Department (READPAST);
SELECT * FROM Sales.SalesPerson;
```

   SQL Server skips (reads past) only one row, and the remaining 15 rows of *Department* are returned, followed by all the *SalesPerson* rows. The READPAST hint is frequently used in conjunction with a TOP clause, in particular TOP 1, where your table is serving as a work queue. Your *SELECT* must get a row containing an order to be processed, but it really doesn't matter which row. So *SELECT* TOP 1 * FROM <OrderTable> returns the first unlocked row, and you can use that as the row to start processing.

4. Open a fourth connection, and execute the following statements:

```
USE AdventureWorks2008 ;
SELECT * FROM HumanResources.Department (READUNCOMMITTED);
SELECT * FROM Sales.SalesPerson;
```

   In this case, SQL Server does not skip anything. It reads all 16 rows from *Department,* but the row for Department 1 shows the dirty data that you changed in step 1. This data has not yet been committed and is subject to being rolled back.

The READUNCOMMITTED hint is probably the least useful because of the availability of row versioning. In fact, anytime you find yourself needing to use this hint, or the equivalent NOLOCK, you should consider whether you can actually afford the cost of one of the snapshot-based isolation levels.

Summary

SQL Server lets you manage multiple users simultaneously and ensure that transactions observe the properties of the chosen isolation level. Locking guards data and the internal resources that make it possible for a multiuser system to operate like a single-user system. You can choose to have your databases and applications use either optimistic or pessimistic concurrency control. With pessimistic concurrency, the locks acquired by data modification operations block users trying to retrieve data. With optimistic concurrency, the locks are ignored and older committed versions of the data are read instead. In this chapter, we looked at the locking mechanisms in SQL Server, including full locking for data and leaf-level index pages and lightweight locking mechanisms for internally used resources. We also looked at the details of how optimistic concurrency avoids blocking on locks and still has access to data.

It is important to understand the issues of lock compatibility and escalation if you want to design and implement high-concurrency applications. You also need to understand the costs and benefits of the two concurrency models.

Index

Symbols and Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

About the Authors

Kalen Delaney

Kalen Delaney has been working with Microsoft SQL Server for over 21 years, and she provides advanced SQL Server training to clients around the world. She has been a SQL Server MVP (Most Valuable Professional) since 1992 and has been writing about SQL Server almost as long. Kalen has spoken at dozens of technical conferences, including every PASS Community Summit held in the United States since the organization's founding in 1999. Kalen is a partner and Director of Training for SQL Tuners (*www.sqltuners.net*), a SQL Server tuning and managed services company based in the northwestern United States.

Kalen is a contributing editor and columnist for *SQL Server Magazine* and the author or co-author of several Microsoft Press books on SQL Server, including *Inside Microsoft SQL Server 7, Inside Microsoft SQL Server 2000, Inside Microsoft SQL Server 2005: The Storage Engine*, and *Inside Microsoft SQL Server 2005: Query Tuning and Optimization*. Kalen blogs at *www.sqlblog.com,* and her personal Web site can be found at *www.SQLServerInternals.com*.

Paul S. Randal

Paul is the managing director of SQLskills.com, which he runs with his wife, Kimberly L. Tripp. He is also a SQL Server MVP and one of the contributing editors of *TechNet Magazine*. Paul joined Microsoft in 1999 after spending five years at DEC working on the OpenVMS file system. He wrote various DBCC commands for SQL Server 2000 and then rewrote all of DBCC CHECKDB for SQL Server 2005 before moving into management on the SQL Server team. During SQL Server 2008 development, he was responsible for the entire Storage Engine.

Paul regularly teaches classes on topics such as database maintenance, high availability, disaster recovery, and SQL Server internals. He is a top-rated presenter at worldwide Tech·Ed and co-chairs the SQL Server Connections conferences. In the last year, Paul has written a large number of SQL Server 2008 materials, including white papers, and articles for *TechNet Magazine*. Paul's popular blog is at *www.SQLskills.com/blogs/paul,* and he can be reached at *Paul@SQLskills.com*.

Kimberly L. Tripp

Kimberly is the president/founder of SQLskills.com, which she started in 1995 after leaving Microsoft, where she held multiple positions, including technical writer for the SQL Server Team and subject matter expert/trainer for Microsoft University. She is a SQL Server MVP, a Microsoft regional director, and a contributing editor of *SQL Server Magazine*. Since 1990, Kimberly has focused on many aspects of SQL Server availability, with emphasis on performance tuning and optimization.

Kimberly regularly teaches classes on topics such as database design, performance tuning, database maintenance, and SQL Server internals. She is a top-rated presenter at worldwide Tech·Ed conferences and the PASS Community Summit, and she co-chairs the SQL Server Connections conferences with Paul Randal. Kimberly has worked with all releases of SQL Server since version 1.0 and has written numerous resources, including online content and webcasts, white papers, and most recently, the Microsoft SQL 2008 JumpStart training class for DBAs. Kimberly's popular blog is at *www.SQLskills.com/blogs/kimberly* and she can be reached at *Kimberly@SQLskills.com*.

Conor Cunningham

Conor Cunningham is principal architect of the SQL Server Core Engine Team, with over 10 years experience building database engines for Microsoft. He specializes in query processing and query optimization, and he designed and/or implemented a number of the query processing features available in SQL Server. Conor holds a number of patents in the field of query optimization, and he has written numerous academic papers on query processing. Conor blogs at "Conor vs. SQL" at *http://blogs.msdn.com/conor_cunningham_msft/default.aspx*.

Adam Machanic

Adam Machanic is a Boston-based independent database consultant, writer, and speaker. He has been involved in dozens of SQL Server implementations for both high-availability OLTP and large-scale data warehouse applications, and he has optimized data access layer performance for several data-intensive applications. Adam has written for numerous Web sites and magazines, including SQLBlog, Simple Talk, Search SQL Server, *SQL Server Professional*, *CoDe*, and *Visual Systems Journal*. He has also contributed to several books on SQL Server, including *Expert SQL Server 2005 Development* (Apress, 2007) and *Inside SQL Server 2005: Query Tuning and Optimization* (Microsoft Press, 2007). Adam regularly speaks at user groups, community events, and conferences on a variety of SQL Server– and .NET-related topics. He is a SQL Server MVP, a Microsoft Certified IT Professional (MCITP), and a member of the INETA North American Speakers Bureau.

Technical Reviewer: Benjamin Nevarez

Ben Nevarez has 15 years of experience with relational databases and has worked with SQL Server since version 6.5. He holds a master's degree in computer science and has been a speaker at several technology conferences, including the PASS Community Summit. Ben is currently a senior database administrator with the American International Group (AIG). When he is not working with SQL Server, Ben spends time with his wife, Rocio, and his three sons, David, Benjamin, and Diego.